Decidable logics combining heap structures and data

P. Madhusudan
University of Illinois at Urbana-Champaign, USA
madhu@illinois.edu

Gennaro Parlato
LIAFA, Paris
gennaro@liafa.jussieu.fr

Xiaokang Qiu
University of Illinois at Urbana-Champaign, USA
qiu2@illinois.edu

Abstract

We define a new logic, STRAND, that allows reasoning with heap-manipulating programs using deductive verification and SMT solvers. STRAND logic ("STRuctures ANd Data" logic) formulas express constraints involving heap structures and the data they contain; they are defined over a class of pointer-structures $\mathcal{R}$ defined using MSO-defined relations over trees, and are of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$, where φ is a monadic second-order logic (MSO) formula with additional quantification that combines structural constraints as well as data-constraints, but where the data-constraints are only allowed to refer to $\vec{x}$ and $\vec{y}$.

The salient aspects of the logic are: (a) the logic is powerful, allowing existential and universal quantification over the nodes, and complex combinations of data-constraints and structural constraints; (b) given a linear block of statements s manipulating the heap and data, and a pre-condition P and a post-condition Q for it written using a Boolean combination of $\exists\vec{x}\varphi$ and $\forall\vec{y}\varphi$ formulas, checking the validity of the associated Hoare-triple $\{P\}s\{Q\}$ reduces to satisfiability of a STRAND formula, and (c) there is a powerful fragment of STRAND for which satisfiability is decidable, where the decision procedure works by combining the theory of MSO over trees and the quantifier-free theory of the underlying data-logic. We demonstrate the effectiveness and practicality of the logic by checking verification conditions generated in proving properties of several heap-manipulating programs, using a tool that combines an MSO decision procedure over trees (MONA) with an SMT solver for integer constraints (Z3).

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords

1. Introduction

A fundamental component of most analysis techniques for complex programs is logical reasoning. The advent of efficient SMT solvers (satisfiability-modulo theory solvers) have significantly advanced the techniques for the analysis of programs. SMT solvers check satisfiability in particular theories (e.g. integers, arrays, theory of uninterpreted functions, etc.), and are often restricted to *quantifier-free* fragments of first-order logic, but support completely automated and efficient decision procedures for satisfiability. Moreover, by using techniques that *combine* theories, larger satisfiable theories can be obtained. In particular, the Nelson-Oppen framework [19] allows generic combinations of *quantifier-free* theories, and has found to be useful in efficient combinations of theories that are implemented by a SAT-solver querying the decision procedures of the component theories.

Satisfiability solvers for theories are tools that advance several analysis techniques. They are useful in test-input generation, where the solver is asked whether there exists an input to a program that will drive it along a particular path; see for example [8]. SMT solvers are used also for static-analysis using abstraction interpretation, where the solver is asked to compute precise abstract transitions, i.e. asked whether there is a concretization of an abstract state a that transitions to a concretization of another abstract state a' (for example see SLAM [2] for predicate abstraction and TVLA [13, 24] for shape-analysis). Solvers are also useful in classical deductive verification, where Hoare-triples that state pre-conditions and post-conditions can be transformed into verification conditions whose validity is checked by the solver; for example BOOGIE use the SMT solver Z3 and ESC/JAVA [7] uses SIMPLIFY to prove verification conditions.

One of the least understood class of theories, however, are theories that combine heap-structure and the data they contain. Analysis of programs that manipulate dynamically allocated memory and perform destructive pointer-updates on this memory, maintaining data-structure invariants (like a binary search tree), require reasoning with heaps with an *unbounded* number of nodes with data stored in them. Reasoning with heap structures and data pose fundamental challenges due to the unboundedness of the data-structures. First, for a logic to be useful, it must be able to place constraints on all parts of the structure (e.g. to say a list is sorted), and hence some form of universal quantification over the heap is absolutely necessary. This immediately rules out classical combinations of theories, like the Nelson-Oppen scheme [19], which cater only to quantifier-free theories. Intuitively, given a constraint on heap structures and data, there may be an *infinite* number of heaps that satisfy the structural constraints, and checking whether any of these heaps can be extended with data to satisfy the constraint cannot be stated over the data-logic (even if it has quantification).

There have been a few breakthroughs in combining heap structures and data recently. For instance, HAVOC [12] supports a logic that ensures decidability using a very highly restrictive syntax, and CSL [5] extends the HAVOC logic mechanism to handle constraints on sizes of structures. However, both these logics have very awkward syntax, that involves the domain being partially ordered with respect to *sorts*, and the logic heavily curtailed so that the decision procedure can move down the sorted structures hierarchically and hence terminate. However, these logics cannot express even some simple properties on trees of unbounded depth, like the property that a tree is a binary search tree. More importantly, the technique for deciding the logic is encoded in the *syntax*, which in turn narrowly aims for a fast reduction to the underlying data-logic, making it hard to extend or generalize.

In this paper, we propose a new fundamental technique for deciding theories that combine heap structures and data. The technique is based on defining a notion of *satisfiability-preserving embeddings* between heap-structures, and extracting the minimal models with respect to these embeddings to synthesize a data-logic formula, which can then be decided by an SMT solver for the data-theory.

The logic* STRAND*: We define a new logic called STRAND (for STructure ANd DAta), that combines a powerful heap-logic with an arbitrary data-logic. STRAND formulas are interpreted over a class of *data-structures* $\mathcal{R}$, and are of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$, where φ is a formula that combines a complete monadic second order logic over the heap-structure (and can have additional quantification), and a data-logic that can constrain the data-fields of the nodes referred to by $\vec{x}$ and $\vec{y}$.

The heap-logic in STRAND is derived from the rich logic tradition of designing decidable monadic second-order logics over graphs, and is extremely expressive in defining structural shapes and invariants. STRAND formulas are interpreted over a recursively defined class of data-structures $\mathcal{R}$, which is defined using a regular set of *skeleton* trees with MSO-defined edge-relations (pointer-relations) between them. This way of recursively defining data-structures is not new, and was pioneered by the PALE system [16], which reasons with purely structural properties of heaps defined in a similar manner. In fact, the notion of *graph types* [10] is a convenient and simple way to define data-structure types and invariants, and is easily translatable to our scheme. Data-structures defined over skeleton trees have enough expressive power to state most data-structure invariants of recursively defined data-structures, including nested lists, threaded trees, cyclic and doubly-linked lists, and separate or loosely connected combinations of these structures. Moreover, they present a class of graphs that have a decidable MSO theory, as MSO on these graphs can be *interpreted* using MSO over trees, which is decidable. In fact, graphs defined this way are one of the largest classes of graphs that have a decidable MSO theory.

As we show in this paper, the STRAND logic is well-suited to reasoning with programs. In particular, assume we are given a (straight-line) program P, a pre-condition on the data-structure expressed as a set of recursive structures $\mathcal{R}$, and a pre-condition and a post-condition expressed in a sub-fragment of STRAND that allows Boolean combination of the existential and universal fragments. We show that checking the invalidity of the associated Hoare-triple reduces to the satisfiability problem of STRAND over a new class of recursive structures $\mathcal{R}'$. This facilitates using STRAND to express a variety of problems, including the applications of test-input generation, finding abstract transitions, and deductive verification described above.

Decidable fragments of* STRAND*: The primary contribution of this paper is in identifying a decidable fragment of STRAND. Our fragment is semantically defined, but syntactically checkable, and works through a notion called *satisfiability-preserving embeddings*. Intuitively, for two heap structures (without data) S and S', S *satisfiability-preservingly embeds* in S' with respect to a STRAND formula ψ if there is an embedding of the nodes of S in S' such that *no matter how the data-logic constraints are interpreted*, if S' satisfies ψ, then so will the submodel S satisfy ψ, by inheriting the data-values. We define the notion of satisfiability-preserving embeddings so that it is entirely structural in nature, and is definable using MSO on an underlying graph that simultaneously represents S, S', and the embedding of S in S'.

If S satisfiability-preservingly embeds in S', then clearly, when checking for satisfiability, we can ignore S' if we check satisfiability for S. More generally, the satisfiability check can be done only for the *minimal* structures with respect to the partial-order (and well-order) defined by satisfiability-preserving embeddings. The decidable fragment STRAND_{dec} is defined to be the class of all formulas for which the set of minimal structures with respect to satisfiability-preserving embeddings is *finite*, and where the quantifier-free theory of the underlying data-logic is decidable.

The decidable fragment of STRAND is semantically defined, but we show that it is syntactically checkable. Given a STRAND formula ψ, we show that we can build a regular finite representation of all the minimal models with respect to satisfiability-preserving embeddings, even if it is an infinite set, using automata-theory. Then, checking whether the number of minimal models is finite is decidable. If the set of minimal models is finite, we show how to enumerate the models, and reduce the problem of checking whether they admit a data-extension that satisfies ψ to a formula in the *quantifier-free* fragment of the underlying data-logic, which can then be decided.

We also exhibit a property on the class of data-structures $\mathcal{R}$, which, if satisfied, admits a decidable *syntactic* fragment of STRAND, for STRAND formulas of the kind $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$, where φ has no additional quantification. For instance, over the class of trees (which is very common, as many nested data-structures without aliasing correspond to trees), if the signature contains only descendent, left-subtree, and right-subtree relations, then this syntactic fragment of STRAND is entirely decidable.

We report also on an implementation of the above decision procedures. For the structural phase, we use MONA [9], a powerful logic for deciding MSO over trees and which, despite its theoretical non-elementary worst-case complexity, works very efficiently on realistic examples, by combining a variety a techniques including tree-automata minimization, BDDs, and guided tree automata. The quantifier-free data-logic we use is the quantifier-free logic of linear arithmetic, and we use the SMT solver Z3 to handle these constraints. We have proved several heap-manipulating programs correct including programs that search and insert into sorted lists, reverse sorted lists, and perform search and insertion into binary-search trees. In each of these cases, the formulas we wrote evaluated to be in the decidable fragment, supporting our thesis that the decidable fragment is natural and useful. The binary search-tree examples that we show correct here *cannot* be proved correct in the frameworks defined by HAVOC and CSL (expressing the binary-search property will violate HAVOC's syntax on sorts).

In summary, we present a general decidability technique for combining heap structures and data, demonstrate syntactically decidable fragments for trees, and present experimental evaluation to show that the decidable combination is expressible and efficiently solvable. We believe that this work breaks new ground in combining structures and data, and the technique may also pave the way for defining decidable fragments of other logics, such as decidable fragments of separation logic that combine structure and data.

2. Motivating examples and logic design

The goal of this section is to present an overview of the issues involved in finding decidable logics that combine heap structure and data, which sets the stage for defining the decidable fragment of the logic STRAND, and motivate the choices of our logic design using simple examples on lists.

Let us consider lists in this section, where each node u has a data-field $d(u)$ that can hold a value (say an integer), and with two variables `head` and `tail` pointing to the first and last nodes of the list, respectively. Consider first-order logic, where we are allowed to quantify over the nodes of the list, and further, for any node x, allowed to refer to the *data-field* of x using the term $d(x)$. Let $x \rightarrow y$ denote that y is the successor of x in the list, and let $x \rightarrow^* y$ denote that x is the same as y or precedes y in the list.

EXAMPLE 2.1. *Consider the formula:*

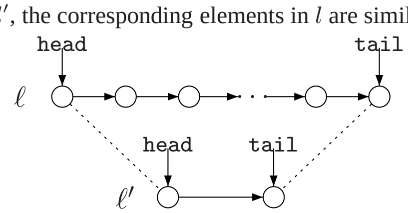
$$\varphi_1: \quad d(\mathtt{head}){=}c_1 \;\wedge\; d(\mathtt{tail}){=}c_2 \;\wedge\; \forall y_1 \forall y_2.((y_1 \rightarrow^* y_2) \Rightarrow d(y_1) \leq d(y_2))$$

The above says that the list must be *sorted* and that the head of the list must have value c_1 and the tail have value c_2. Note that the formula is satisfiable iff $c_1 \leq c_2$, and in which case it is actually satisfied by a list containing *just two elements*, pointed to by `head` and `tail`, with values c_1 and c_2, respectively.

In fact, the property that the formula is satisfiable by a two-element list has *nothing* really to do with the data-constraints involved in the above formula. Assume that we have no idea as to what the data-constraints mean, and hence look upon the above formula by replacing all the data-constraints using uninterpreted predicates $p_1, p_2, \ldots$ to get the formula:

$$\begin{aligned}\widehat{\varphi}_1 : \quad & p_1(\texttt{head}) \ \wedge p_2(\texttt{tail}) \ \wedge \\ & \forall y_1 \forall y_2.((y_1 \rightarrow^* y_2) \Rightarrow p_3(y_1, y_2))\end{aligned}$$

Now, we do not know whether the formula is satisfiable (for example, p_1 may be unsatisfiable). But we still do know that two-element lists are *always sufficient*. In other words, *if there is a list that satisfies the above formula, then there is a two element list that satisfies it*. The argument is simple: take any list l that satisfies the formula, and form a new list l' that has only the head and tail of the list l, with an edge from head to tail, and with data values inherited from l (see figure below). It is easy to see that l' satisfies the formula as well, since whenever two nodes are related by $\rightarrow^*$ in the list l', the corresponding elements in l are similarly related.



This property, of course, does not hold on all formulas, as we see in the example below.

EXAMPLE 2.2. *Consider the formula:*

$$\begin{aligned}\varphi_2 : \quad & d(\texttt{head}){=}c_1 \ \wedge \ d(\texttt{tail}){=}c_2 \ \wedge \\ & \forall y_1 \forall y_2.((y_1 \rightarrow y_2) \Rightarrow d(y_2) = d(y_1) + 1))\end{aligned}$$

The above says that the values in the list increase by one as we go one element down the list, and that the head and tail of the list have values c_1 and c_2, respectively. This formula is satisfiable iff $c_1 < c_2$. However, there is no bound on the size of the minimal model that is *independent* of the data-constraints. For example, if $c_1 = 1$ and $c_2 = 10^6$, then the smallest list that satisfies the formula has a million nodes. In other words, the data-constraints place arbitrary lower bounds on the *size* of the minimal structure that satisfies the formula.

Intuitively, the formula φ_2 refers to *successive* elements in the list, and hence a large model that satisfies the formula is not necessarily contractible to a smaller model. The formula φ_1 in the sortedness example (Example 2.1) referred to pairs of elements that were reachable, leading to contraction of large models to small ones.

Recall that the design principle of the decidable fragment of STRAND is to examine the structural constraints in a formula φ, and enumerate a *finite* set of structures such that the formula is satisfiable iff it one of these structures can be populated with values to satisfy the formula. This strategy necessarily fails for the above formula φ_2, as there is no class of finite structures that adequately captures all models of the formula, independent of the data-constraints. The *sortedness* formula φ_1 in the first example is part of the decidable fragment of STRAND, while φ_2 is outside of it.

EXAMPLE 2.3. *Consider the formula:*

$$\begin{aligned}\varphi_3 : \quad & d(\texttt{head}){=}c_1 \ \wedge \ d(\texttt{tail}){=}c_2 \ \wedge \\ & \forall y_1((y_1 \neq \texttt{tail}) \Rightarrow \exists y_2.(d(y_2) = d(y_1) + 1))\end{aligned}$$

This formula says that for any node n except the tail, there is some node n' that has the value $d(n) + 1$. Notice that the formula is satisfiable if $c_1 < c_2$, but still there is no a priori bound on the minimal model that is independent of the data-constraints. In particular, if $c_1 = 0$ and $c_2 = 10^6$, then the smallest model is a list with 10^6 nodes. Moreover, the reason why the bounded structure property fails is not because of the data-constraints referring to successive elements as in Example 2, but rather because the above formula has a $\forall\exists$ prefix quantification of data-variables. Formulas where an existential quantification follows a universal quantification in the prefix seldom have bounded models, and STRAND hence only allows formulas with $\exists^*\forall^*$ quantification prefixes. Note that quantification of *structure variables* (variables that quantify over nodes but whose data-field is not referenced in the formula) can be arbitrary, and in fact we allow STRAND formulas to even have *set quantification* over nodes.

The Bernays-Schönfinkel-Ramsey class: Having motivated formulas with the $\exists^*\forall^*$ quantification, it is worthwhile to examine this fragment in classical first-order logic (over arbitrary infinite universes), which is known as the Bernays-Schönfinkel-Ramsey class, and is a classical decidable fragment of first-order logic [4].

Consider first a purely relational vocabulary (assume there are no functions and even no constants). Then, given a formula $\exists\vec{x}\forall\vec{y}\varphi(\vec{x}, \vec{y})$, let M be a model that satisfies this formula. Let v be an interpretation for $\vec{x}$ such that M under v satisfies $\forall\vec{y}\varphi(\vec{x}, \vec{y})$. Then it is not hard to argue that the submodel obtained by picking only the elements used in the interpretation of $\vec{x}$ (i.e. $v(\vec{x})$), and projecting each relation to this smaller set, satisfies the formula $\exists\vec{x}\forall\vec{y}\varphi(\vec{x}, \vec{y})$ as well [4]. Hence a model of size at most k always exists that satisfies φ, if the formula is satisfiable, where k is the size of the vector of existentially quantified variables $\vec{x}$. This *bounded model property* extends to when constants are present as well (the submodel should include all the constants) but fails when more than two functions are present. Satisfiability hence reduces to propositional satisfiability, and this class is also called the *effectively propositional* class, and SMT solving for this class exists.

The decidable fragment of STRAND is fashioned after a similar but more complex argument. Given a subset of nodes of a model, the subset itself may not form a valid graph/data-structure. We define a notion of *submodels* that allows us to extract proper subgraphs that contain certain nodes of the model. However, the relations (edges) in the submodel will *not* be the projection of edges in the larger model. Consequently, the submodel may not satisfy a formula, even though the larger model does.

We define a notion called *satisfiability-preserving* embeddings that allows us to identify when a submodel S of T is such that, whenever T satisfies ψ under some interpretation of the data-logic, S can *inherit* values from T to satisfy ψ as well. This is considerably more complex and is the main technical contribution of the paper. We then build decision procedures to check the minimal models according to this embedding relation.

3. Recursive data-structures

We now define recursive data-structures using a formalism that defines the nodes and edges using MSO formulas over a regular set of trees. Intuitively, a set of data-structures is defined by taking a regular class of trees that act as a *skeleton* over which the data-structure will be defined. The precise set of nodes of the tree that correspond to the nodes of the data-structure, and the edges between these nodes (which model pointer fields) will be captured using MSO formulas over these trees. We call such classes of data-structures *recursively definable data-structures*.

Recursively definable data-structures are very powerful mechanisms for defining invariants of data-structures. The notion of *graph types* [10] is a very similar notion, where again data-structure invariants are defined using a tree-backbone but where edges are defined using *regular path expressions*. Graph types can be modeled directly in our framework; in fact, our formalism is more powerful.

The framework of recursively definable data-structures are also interesting because they define classes of graphs that have a *decidable monadic second-order theory*. In other words, given a class $\mathcal{C}$ of recursively defined data-structures, the satisfiability problem for MSO formulas over $\mathcal{C}$ (i.e. the problem of checking, given φ, whether there is some structure $R \in \mathcal{C}$ that satisfies φ) is decidable. The decision procedure works by interpreting the MSO formula on the tree-backbone of the structures. In fact, our framework can capture all graphs definable using *edge-replacement grammars*, which are one of the most powerful classes of graphs known that have a decidable MSO theory [6].

3.1 Graphs and monadic second-order logics

A labeled (directed) graph G over a finite set of vertex-labels L_v and a finite set of edge labels L_e is a 6-tuple, $G = (V, E, \mu, \nu, L_v, L_e)$, where V is the set of nodes, $E \subseteq V \times V$ is a set of edges, $\mu : V \to 2^{L_v}$ that assigns a subset of labels to each vertex, and $\nu : E \to 2^{L_e}$ that assigns a subset of labels each edge.

Monadic second-order logic (MSO) on graphs over the labels (L_v, L_e) is the standard MSO on structures of the form $(U, E, \{Q_a\}_{a\in L_v}, \{E_b\}_{b\in L_e})$ where U represents the universe, E is a binary relation capturing the edge relation, Q_a is a monadic predicate that captures all nodes whose labels contain a, and E_b is a binary relation that captures all edges whose label contain b (note that $E_b \subseteq E$, for every $b \in L_e$). However, we also allow Boolean variables and quantification over them[1].

Let us fix a countable set of first-order variables FV (first-order variables will be denoted by s, t, etc.) and a countable set of set variables SV (set-variables will be denoted by S, T, etc.). Let us also fix a countable set of Boolean variables BV(denoted by p, q, etc.) The syntax of the logic is:

$$\varphi ::= p \mid Q_a(s) \mid E(s,t) \mid E_b(s,t) \mid s = t \mid s \in S \mid$$

$$\varphi \vee \varphi \mid \neg\varphi \mid \exists s.\varphi \mid \exists S.\varphi$$

where $a \in L_v$, $b \in L_e$, $s,t \in FV$, $S \in SV$, and $p \in BV$.

3.2 Recursively defined data-structures

Let Σ be a finite alphabet. For any $k \in \mathbb{N}$, let $[k]$ denote the set $\{1, \ldots k\}$.

A k-ary Σ-labeled tree is a pair (V, λ), where $V \subseteq [k]^*$, and V is non-empty and prefix-closed, and $\lambda : V \to \Sigma$. The edges of the tree are implicitly defined: that is $u.i$ is the i'th child of u, for every $u, u.i \in V$, where $u \in [k]^*$ and $i \in [k]$. Trees are seen as graphs with Σ-labeled vertices and edge relations $E_i(x,y)$ that define the i'th-child edges. Monadic second-order logic over trees is the MSO logic over these graphs.

Formally, we define classes of recursively defined data-structures as follows.

DEFINITION 3.1. *A class $\mathcal{C}$ of recursively defined data-structures is specified by a tuple $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a\in L_v}, \{\beta_b\}_{b\in L_e})$, where ψ_{Tr} is an MSO sentence, ψ_U is a unary predicate defined in MSO, and each α_a and β_b are monadic and binary predicates defined using MSO, where all MSO formulas are over k-ary trees, for some $k \in \mathbb{N}$, .* ∎

[1] Classical definitions of MSO do not allow such Boolean quantification, but we will find it useful in our setting. These variables can be easily removed; e.g. instead of quantifying over a Boolean variable p, we can quantify over a set X and convert every occurrence of p to a formula that expresses that X is empty.

Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a\in L_v}, \{\beta_b\}_{b\in L_e})$ and T be a k-ary Σ-labeled tree. Then $T = (V, E)$ defines a graph $Graph(T) = (N, E, \mu, \nu, L_v, L_e)$ defined as follows:

- $N = \{s \in V \mid \psi_U(s) \text{ holds in } T\}$
- $E = \{(s, s') \mid \beta_b(s, s') \text{ holds in } T \text{ for some } b \in L_e\}$
- $\mu(s) = \{a \in L_v \mid \alpha_a(s) \text{ holds in } T\}$
- $\nu((s, s')) = \{b \in L_e \mid \beta_b(s, s') \text{ holds in } T\}$.

The class of graphs defined by $\mathcal{R}$ is the set $Graph(\mathcal{R}) = \{Graph(T) \mid T \models \psi_{Tr}\}$.

EXAMPLE 3.2. *Let us define a class of recursive data-structures that consists of trees where the leaves of the tree are connected by a linked list. The class of trees will be the class of binary trees (with edges E_1 and E_2 representing left- and right-child relations), and we define the $next$-edge relation for the list using an MSO predicate:*

$$E_{next}(s,t) = leaf(s) \wedge leaf(t) \wedge \exists z_1, z_2, z_3 (E_1(z_3, z_1) \wedge E_2(z_3, z_2) \wedge RightMostPath(z_1, s) \wedge LeftMostPath(z_2, t))$$

where $leaf(x)$ is a subformula that checks if x is a leaf, and $RightMostPath(x,y)$ (and $LeftMostPath(x,y)$) is a formula that checks if y is in the right-most (left-most, respectively) path from x.

4. STRAND: A logic over heap structures and data

4.1 Definition

We now introduce our logic STRAND ("STRucture ANd Data"). STRAND is a two-sorted logic interpreted on program heaps with both locations and their carried data. Given a first order theory $\mathcal{D}$ of sort $Data$, and given $\mathcal{L}$, a monadic second order (MSO) theory over (L_v, L_e)-labeled graphs, of sort Loc, the syntax of STRAND is presented in Figure 1. STRAND is defined over the two-sorted signature $\Gamma(\mathcal{D}, \mathcal{L}) = Sig(\mathcal{D}) \cup Sig(\mathcal{L}) \cup \{data\}$, where $data$ is a function of sort $Loc \to Data$. STRAND formulas are of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are ∃DVar and ∀DVar, respectively, of sort Loc (we also refer both as DVar), ψ is an MSO formula with atomic formulas of the form either $\gamma(e_1, \ldots, e_n)$ or $\alpha(v_1, \ldots, v_n)$. $\gamma(e_1, \ldots, e_n)$ is an atomic $\mathcal{D}$-formula in which the data carried by Loc-variables can be referred as $data(x)$ or $data(y)$. $\alpha(v_1, \ldots, v_n)$ is just an atomic formula from $\mathcal{L}$. Note that additional variables are allowed in $\varphi(\vec{x}, \vec{y})$, both first order or second order, but $\gamma(e_1, \ldots, e_n)$ is only allowed to refer to $\vec{x}$ and $\vec{y}$.

A model for STRAND is a structure $\mathcal{M} = \langle \mathcal{M}_{Loc}, \mathcal{M}_{Data}, M_{map}\rangle$. $\mathcal{M}_{Loc}$ is an $\mathcal{L}$-model (i.e. a labeled graph) with M_{Loc} as the underlying set of nodes, and $\mathcal{M}_{Data}$ is a $\mathcal{D}$-model with M_{Data} as the underlying set. M_{map} is an interpretation for the function $data$ of sort $M_{Loc} \to M_{Data}$. The semantics of STRAND formulas is the natural extension of the logics $\mathcal{L}$ and $\mathcal{D}$.

We will refer to a $\mathcal{L}$-model as a *graph-model*. The *data-extension* of graph model $\mathcal{M}_{Loc}$ is a STRAND model $\langle \mathcal{M}_{Loc}, \mathcal{M}_{Data}, M_{map}\rangle$.

Undecidability. STRAND is an expressive logic, as we will show below, but it is undecidable in general, even if both its underlying theories $\mathcal{D}$ and $\mathcal{L}$ are decidable. Let $\mathcal{D}$ be linear integer arithmetic and $\mathcal{L}$ be the standard MSO logic over lists. It is easy to model an execution of a 2-counter machine using a list with integers. Each configuration is represented by two adjacent nodes, which are labeled by the current instruction. The data fields of the two nodes hold the value of the two registers, respectively. Then a halting computation can be expressed by a STRAND formula. Hence the satisfiability of the STRAND logic is undecidable, though the underlying logics $\mathcal{L}$ and $\mathcal{D}$ are decidable.

$\exists$DVar	x	$\in$	Loc
$\forall$DVar	y	$\in$	Loc
GVar	z	$\in$	Loc
Variable	v	$::=$	$x \mid y \mid z$
Set − Variable	S	$\in$	2^{Loc}
Constant	c	$\in$	$Sig(\mathcal{D})$
Function	g	$\in$	$Sig(\mathcal{D})$
$\mathcal{D}-$Relation	γ	$\in$	$Sig(\mathcal{D})$
$\mathcal{L}-$Relation	α	$\in$	$Sig(\mathcal{L})$
Expression	e	$::=$	$data(x) \mid data(y) \mid c \mid g(e_1, \ldots, e_n)$
AFormula	φ	$::=$	$\gamma(e_1, \ldots, e_n) \mid \alpha(v_1, \ldots, v_n)$
			$\mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$
			$\mid \exists z.\varphi \mid \forall z.\varphi \mid \exists S.\varphi \mid \forall S.\varphi$
$\forall$Formula	ω	$::=$	$\varphi \mid \forall y.\omega$
Formula	ψ	$::=$	$\omega \mid \exists x.\psi$

Figure 1. Syntax of STRAND

4.2 Examples

We now show various examples to illustrate the expressiveness of STRAND. We sometimes use $d()$ instead of $data()$, for brevity.

EXAMPLE 4.1 (sorted list). *We first revisit the motivating Example 2.1 presented in Section 2. In the formula φ_1, y_1 and y_2 must be DVar since their data fields are referred. Thus φ_1 can be rewritten in* STRAND *as*

$$\psi_{sorted} \equiv \forall y_1 \forall y_2.(\ d(\texttt{head}){=}c_1\ \wedge\ d(\texttt{tail}){=}c_2 \wedge$$
$$((y_1 \rightarrow^* y_2) \Rightarrow d(y_1) \leq d(y_2))\ )$$

EXAMPLE 4.2 (Binary search tree). *In* STRAND, *a* binary search tree *(BST) is interpreted as a binary tree data structure with an additional* key *field for each node. The keys in a BST are always stored in such a way as to satisfy the binary-search-tree property, expressed in* STRAND *as follows:*

$$\mathit{leftsubtree}(y_1, y_2) \equiv \exists z(\mathit{left}(y_1, z) \wedge z \rightarrow^* y_2)$$

$$\mathit{rightsubtree}(y_1, y_2) \equiv \exists z(\mathit{right}(y_1, z) \wedge z \rightarrow^* y_2)$$

$$\psi_{bst} \equiv \forall y_1 \forall y_2(\ (\mathit{leftsubtree}(y_1, y_2) \Rightarrow d(y_2) < d(y_1)) \wedge$$
$$((\mathit{rightsubtree}(y_1, y_2) \Rightarrow d(y_1) \geq d(y_2))\ )$$

Note that ψ_{bst} has an existentially quantified variable z in GVar after the universal quantification of y_1, y_2. However, as z is a structural quantification (whose data-field cannot be referred to), this formula is in STRAND.

EXAMPLE 4.3 (Two disjoint lists). *In* separation logic*[23], a novel binary operator* $*$, *or* separating conjunction, *is defined to assert that the heap can be split into two disjoint parts where its two arguments hold, respectively. Such an operator is useful in reasoning with* frame conditions *in program verification. Thanks to the powerful expressiveness of MSO logic, the separating conjunction is also expressible in* STRAND. *For example,* $(\texttt{head}_1 \rightarrow^* \texttt{tail}_1) * (\texttt{head}_2 \rightarrow^* \texttt{tail}_2)$ *states, in separation logic, that there are two disjoint lists such that one list is from* $\texttt{head}_1$ *to* $\texttt{tail}_1$, *and the other is from* $\texttt{head}_2$ *to* $\texttt{tail}_2$. *This formula can be written in* STRAND *as:*
$\exists S_1 \exists S_2(\mathit{disjoint}(S_1, S_2)\ \wedge \texttt{head}_1{\in}S_1 \wedge \texttt{tail}_1{\in}S_1 \wedge$
$\texttt{head}_2{\in}\ S_2{\wedge}\, \texttt{tail}_2{\in}S_2\ \wedge \texttt{head}_1 \rightarrow^* \texttt{tail}_1 \wedge \texttt{head}_2 \rightarrow^* \texttt{tail}_2)$
where $\mathit{disjoint}(S_1, S_2) = \nexists z(z \in S_1 \wedge z \in S_2)$

5. Deciding STRAND fragments

5.1 Removing existential quantification:

Given a STRAND formula $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$ over a class of recursively defined data-structures $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$, we can transform this to an *equisatisfiable* formula $\forall\vec{x}\forall\vec{y}\varphi'(\vec{x},\vec{y})$ over a *different* class of recursive data-structures $\mathcal{R}'$, where data-structures in $\mathcal{R}'$ are data-structures in $\mathcal{R}$ with a new unary predicate that gives a valuation for the variables in $\vec{x}$. We won't define this formally, but this is an easy transformation: we modify ψ_{Tr} to accept trees with extra labelings a_i that give (an arbitrary) singleton valuation of each $x_i \in \vec{x}$ that satisfies ψ_U, and introduce new unary predicates $Val_i(x) = Q_{a_i}(x)$, and define $\varphi'(\vec{x},\vec{y})$ to be $(\wedge_i Val_i(x_i)) \Rightarrow \varphi(\vec{x},\vec{y})$. It is easy to see there is a graph in $\mathit{Graph}(\mathcal{R})$ that satisfies $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$ iff there is a graph in $\mathit{Graph}(\mathcal{R}')$ that satisfies $\forall\vec{x}\forall\vec{y}\varphi'(\vec{x},\vec{y})$. The latter is a STRAND formula with no existential quantification of variables whose data is referred to by the formula. Let us refer to these formulas with no leading existential quantification on data-variables as *universal* STRAND *formulas*; we will now outline techniques to solve the satisfiability problem of a certain class of universal STRAND formulas.

5.2 Submodels

Let us fix a class of recursively defined data-structures $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ for the rest of this section.

We first need to define the notion of *submodels* of a model. The definition of a submodel will depend on the particular class of recursively defined data-structures we are working with, since we want to exploit the tree-representation of the models, which in turn will play a crucial role in deciding fragments of STRAND, as it will allow us to check satisfiability-preserving embeddings. In fact, we will define the submodel relation between trees that satisfy ψ_{Tr}.

Let $T = (V, \lambda)$ be a tree that satisfies ψ_{Tr}, and let $S \subseteq V$. Then we say that S is *a valid subset* of V if the following hold:

- S is non-empty, and least-ancestor closed (i.e. for any $s, s' \in S$, the least common ancestor of s and s' in T also belongs to S).
- The *subtree* defined by S, denoted $\mathit{Subtree}(T, S)$, is the tree with nodes S, and where the i'th child of a node $u \in S$ is the (unique) node $u' \in S$ closest to u that is in the subtree rooted at the i'th child of u. (This is uniquely defined since S is least-ancestor closed.) Then we require that $\mathit{Subtree}(T, S)$ also satisfies ψ_{Tr}.
- We also require that for every $s \in S$, if $\psi_U(s)$ holds in $\mathit{Subtree}(T, S)$, then $\psi_U(s)$ holds in T as well.

A tree $T' = (V', \lambda')$ is said to be a *submodel* of $T = (V, \lambda)$ if there is a valid subset S of V such that T' is isomorphic to $\mathit{Subtree}(T, S)$. Note that in this case $\mathit{Graph}(T')$ necessarily satisfies ψ_{Tr} and hence is a valid data-structure.

Intuitively, $T' = (V', \lambda')$ is a *submodel* of $T = (V, \lambda)$ if the vertices of T' can be embedded in T, preserving the tree-structure. The nodes of the $\mathit{Graph}(T')$, are a subset of the nodes of $\mathit{Graph}(T)$ (because of the last condition in the definition of a submodel), and, given a valid subset S, there is in fact an injective mapping from the nodes of $\mathit{Graph}(T')$ to $\mathit{Graph}(T)$. For technical convenience, we will work with valid subsets mostly, as fixing the precise embedding helps in the decision procedures.

5.3 Structural abstractions of STRAND formulas

Let $\psi = \forall\vec{y}\ \varphi(\vec{y})$ be a universal STRAND formula.

We now define the *structural abstraction* of ψ as follows. Let $\gamma_1, \gamma_2, \ldots, \gamma_r$ be the *atomic* relational formulas of the data-logic in φ. Note that each of these relational formulas will be over the data fields of variables in $\vec{y}$ only (since the data-logic is restricted to working over the terms $data(y)$, where $y \in \vec{y}$).

Consider evaluating ψ over a particular model. After fixing a particular valuation of $\vec{y}$, notice that the data-relations γ_i get all fixed, and evaluate to true or false. Moreover, once the values of γ_i are fixed, the rest of the formula is purely *structural* in nature. Now, if ψ is to hold in the model, then no matter how we choose to evaluate $\vec{y}$ over the nodes of the model, the γ_i relations must evaluate to true or false in such a way that φ holds.

Since we want, in the first phase, to *ignore* the data-constraints entirely, we will abstract ψ using a purely structural formula by using Boolean variables $b_1, \ldots b_r$ instead of the data-relations $\gamma_1, \gamma_2, \ldots, \gamma_r$. However, since these Boolean variables get determined only *after* the valuation of $\vec{y}$ gets determined, and since we are solving for satisfiability, we existentially quantify over these Boolean variables and quantify them *after* the quantification of $\vec{y}$. Formally[2],

DEFINITION 5.1. *Let $\psi = \forall \vec{y}\ \varphi(\vec{y})$ be a universal* STRAND *formula, and let the atomic relational formulas of the data-logic that occur in φ be $\gamma_1, \gamma_2, \ldots, \gamma_r$. Then its structural abstraction $\widehat{\psi}$ is defined as the pure MSO formula on graphs*

$$\forall \vec{y}\ \exists b_1 \ldots b_r\ \varphi'(\vec{y}, \vec{b})$$

where φ' is φ but where every occurrence of γ_i is replaced with b_i.

For example, consider the sortedness formula ψ_{sorted} from Example 4.1 in Section 4. Then

$$\begin{aligned}\widehat{\psi}_{sorted}: \quad & \forall y_1 \forall y_2.\ \exists b_1\ (d(\mathtt{head}){=}c_1 \ \wedge \ d(\mathtt{tail}){=}c_2 \ \wedge \\ & ((y_1 \rightarrow^* y_2) \Rightarrow b_1)\end{aligned}$$

Note that each Boolean variable b_i replaces an atomic relational formula γ_i, where γ_i places some data-constraint on the data-fields of some of the universally quantified variables. The structural abstraction has not only lost the constraint, but has even lost the precise variables whose data-fields the constraint was over. Nevertheless, the abstraction is enough to define satisfiability-preserving embeddings.

The following proposition is obvious; it says that if a universal STRAND formula ψ is satisfiable, then so is its structural abstraction $\widehat{\psi}$. The proposition is true because the values for the Boolean variables can be set in the structural abstraction precisely according to how the relational formulas γ_i evaluate in ψ:

PROPOSITION 5.2. *Let $\psi = \forall \vec{y} \varphi(\vec{y})$ be a universal* STRAND *formula, and $\widehat{\psi}$ be its structural abstraction. Then, if ψ is satisfiable over a set of recursive data-structures $\mathcal{R}$, then the MSO formula on graphs (with no constraints on data) $\widehat{\psi}$ is also satisfiable over $\mathcal{R}$.*

5.4 Satisfiability-preserving embeddings

We are now ready to define satisfiability-preserving embeddings using structural abstractions. Given a model defined by a tree $T = (V, \lambda)$ satisfying ψ_{Tr}, and a valid subset $S \subseteq V$, and a universal STRAND formula ψ, we would like to define the notion of when the submodel defined by S satisfiability-preservingly embeds in the model. The most crucial requirement for the definition is that if S satisfiability-preservingly embeds in T, then we require that if there is a data-extension of $Graph(T)$ to satisfy ψ, then the nodes of the submodel defined by S, $Graph(S)$, can inherit the data-values and also satisfy ψ. The notion of structural abstractions defined above allows us to define such a notion.

Intuitively, if a model satisfies ψ, then it would satisfy $\widehat{\psi}$ too, as for every valuation of $\vec{y}$, there is some way it would satisfy the atomic data-relations, and using this we can pull out a valuation for the Boolean variables to satisfy $\widehat{\psi}$ (as in the proof of Proposition 5.2 above). Now, since the data-values in the submodel are *inherited* from the larger model, the atomic data-relations would hold *in the same way* as they do in the larger model. However, the submodel may not satisfy ψ if the conditions on the truth- and false-hood of these atomic relations demanded by ψ are not the same.

For example, consider a list and a sublist of it. Consider a formula that demands that for any two successor elements y_1, y_2 in the list, the data-value of y_2 is the data-value of y_1 incremented by 1 (as in the *successor* example in Section 2):

$$\psi \equiv \quad \forall y_1 \forall y_2.(\ (y_1 \rightarrow y_2) \Rightarrow (d(y_2) = d(y_1) + 1)\ )$$

Now consider two nodes y_1 and y_2 that are successors in the sublist but not successors in the list. The list hence could satisfy the formula by setting the data-relation $\gamma : d(y_2) = d(y_1) + 1$ to false. Since the sublist inherits the values, this would be false in the sublist as well, but the sublist will *not* satisfy the formula. We hence want to ensure that *no matter how the larger model satisfies the formula* using some valuation of the atomic data-relations, the submodel will be able to satisfy the formula using the *same valuation of the atomic data-relations*. This leads us to the following definition:

DEFINITION 5.3. *Let $\psi = \forall \vec{y}\ \varphi(\vec{y})$ be universal* STRAND *formula, and let its structural abstraction be $\widehat{\psi} = \forall \vec{y}\ \exists \vec{b}\ \varphi'(\vec{y}, \vec{b})$. Let $T = (V, \lambda)$ be a tree that satisfies ψ_{Tr}, and let a submodel be defined by $S \subseteq V$. Then S is said to* satisfiability-preservingly embed *into T wrt ψ if for every possible valuation of $\vec{y}$ over the elements of S, and for every possible Boolean valuation of $\vec{b}$, if $\varphi'(\vec{y}, \vec{b})$ holds in the graph defined by T under this valuation, then the submodel defined by S, $Graph(S)$, also satisfies $\varphi'(\vec{y}, \vec{b})$ under the same valuation.*

Let us also say that S *strictly satisfiability-preservingly embeds* into T if S satisfiability-preservingly embeds into T and $S \neq T$.

The satisfiability-preserving embedding relation can be seen as a *partial* order over trees (a tree T' satisfiability-preservingly embeds into T if there is a subset S of T such that S satisfiability-preservingly embeds into T and $Subtree(T, S)$ is isomorphic to T'); it is easy to see that this relation is reflexive, anti-symmetric and transitive.

It is now not hard to see that if S satisfiability-preservingly embeds into T wrt ψ, and $Graph(T)$ satisfies ψ, then $Graph(S)$ also necessarily satisfies ψ, which is the main theorem we seek.

THEOREM 5.4. *Let $\psi = \forall \vec{y} \varphi(\vec{y}$ be universal* STRAND *formula. Let $T = (V, \lambda)$ be a tree that satisfies ψ_{Tr}, and S be a valid subset of T that satisfiability-preservingly embeds into T wrt ψ. Then, if there is a data-extension of $Graph(T)$ that satisfies ψ, then there is a data-extension of $Graph(Subtree(T, S))$ that satisfies ψ.*

Proof The gist of the proof of the above theorem goes similar to the arguments given above. Consider a data-extension of $Graph(T)$ that satisfies ψ. Each node u of $Graph(Subtree(T, S))$ corresponds to a unique node u' in $Graph(T)$ (see above). Define the data-extension of $Graph(Subtree(T, S))S$ by assigning the data-value of each node u to the data-value of the corresponding node u' in $Graph(T)$. For any valuation of $\vec{y}$ over $Graph(Subtree(T, S))$,

[2] The definition of structural abstractions can be strengthened in two ways. First, if γ_i and γ_j are of the same arity and over $\vec{z}$ and $\vec{z'}$, respectively, and further uniformly replacing z_i with z'_i in γ_i yields γ', then we can express the constraint $((\vec{z_i} = \vec{z_i}') \Rightarrow (b_i \Leftrightarrow b_j))$, in the inner formula φ'. Second, if a constraint γ_i involves only existentially quantified variables in $\vec{x}$, then we can move the quantification of b_i outside the universal quantification. Doing these steps gives a more accurate structural abstraction, and in practice, restricts the number of models created. We use these more precise abstractions in the experiments, but use the less precise abstractions in the theoretical narrative. The proofs in this section, however, smoothly extend to the more precise abstractions.

consider the corresponding valuation over T. Since the data-extension of $Graph(T)$ satisfies φ, by Proposition 5.2, $Graph(T)$ must satisfy the formula φ with its atomic data-predicates replaced by Boolean variables. Since S satisfiability-preservingly embeds in T, the same valuation of the Boolean variables also satisfies φ with its atomic data-predicates replaced by Boolean variables. Since the data-extension of $Graph(Subtree(T, S))$ is derived by inheriting the data-values from the data-extension of $Graph(T)$, it follows that the data-values of $\vec{y}$ satisfy the same predicates γ_i as they did in the data-extension of $Graph(T)$. Hence it follows that $\varphi(\vec{y})$ holds in the extension of $Graph(Subtree(T, S))$ as well. Since this is true for any valuation of $\vec{y}$ over S, it follows that the data-extension of $Graph(Subtree(T, S))$ satisfies ψ. ∎

Notice that the above theorem crucially depends on the formula being universal over data-variables. For example, if the formula was of the form $\forall y_1 \exists y_2 \gamma(y_1, y_2)$, then we would have no way of knowing *which* nodes are used for y_2 in the data-extension of $Graph(T)$ to satisfy the formula. Without knowing the precise meaning of the data-predicates, we would not be able to declare that whenever a data-extension of $Graph(T)$ is satisfiable, a data-extension of a strict submodel S is satisfiable (even over lists).

The above notion of *satisfiability preserving embeddings* is the property that will be used to decide if a formula falls into our decidable fragment.

5.5 STRAND$_{dec}$: A decidable fragment of STRAND

We are now ready to define STRAND$_{dec}$, the most general decidable fragment of STRAND in this paper. This fragment is semantically defined (but syntactically checkable, as we show below), and intuitively contains all STRAND formulas which have a *finite number of minimal models* with respect to the partial-order defined by satisfiability-preserving embeddings.

Formally, let $\psi = \forall \vec{y} \varphi(\vec{y})$ be universal STRAND formula, and let $T = (V, \lambda)$ be a tree that satisfies ψ_{Tr}. Then we say that T is *a minimal model* with respect to ψ if there is no strict subset S of T that is a valid subset of V and satisfiability-preservingly embeds in T.

DEFINITION 5.5. *Let $\mathcal{R}$ be a recursively defined set of data-structures.*

A universal formula $\psi = \forall \vec{y}\ \varphi(\vec{y})$ is in STRAND$_{dec}$ *iff the number of minimal models with respect to $\mathcal{R}$ and ψ is finite.*

A STRAND *formula of the form $\psi = \exists \vec{x}\ \forall \vec{y}\ \varphi(\vec{x}, \vec{y})$ is in* STRAND$_{dec}$ *iff the corresponding equi-satisfiable universal formula ψ' over set of data-structure $\mathcal{R}'$ (as defined above) is in* STRAND$_{dec}$. ∎

We now show that we can *effectively check* if a STRAND belongs to the decidable fragment STRAND$_{dec}$. The idea, intuitively, is to express that a model is a minimal model with respect to satisfiability-preserving embeddings, and then check, using automata theory, that the number of minimal models is finite.

Let $\psi = \forall \vec{y} \varphi(\vec{y})$ be universal STRAND formula, and let its structural abstraction be $\widehat{\psi} = \forall \vec{y} \exists \vec{b} \varphi'(\vec{y}, \vec{b})$.

We now show that we can define an MSO formula $MinModel$, such that for any tree $T = (V, \lambda)$, $MinModel$ holds in T iff T defines a minimal model with respect to satisfiability-preserving embeddings.

Before we do that, we need some technical results and notation. Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$.

We first show that any (pure) MSO formula δ on (L_v, L_e)-labeled graphs that correspond to submodels can be *interpreted* on trees. More formally, we show a (pure) MSO formula δ on (L_v, L_e)-labeled graphs can be transformed syntactically to a (pure) MSO formula δ' such that for any tree $T = (V, \lambda)$ $Graph(Subtree(T, S))$ satisfies δ iff T satisfies δ'.

This is not hard to do, since the graph is defined using MSO formulas on the trees, and we can adapt these definitions to work over the tree instead. In fact, this is the reason why MSO on recursive data-structures is decidable: we can translate the formula to trees, and check satisfiability of the transformed formula over trees that satisfy ψ_{Tr}. The transformation is given by the following function $interpret$; the predicates for edges, and the predicates that check vertex labels and edges labels are transformed according to their definition, and all quantified variables are restricted to quantify over nodes that satisfy ψ_U.

- $interpret(p) = p$
- $interpret(Q_a(s)) = \alpha_a(s)$, for every $a \in L_v$
- $interpret(E(s,t)) = \bigvee_{b \in L_e} \beta_b(s,t)$
- $interpret(E_b(s,t)) = \beta_b(s,t)$, for every $b \in L_e$
- $interpret(s = t) = (s = t)$
- $interpret(s \in W) = s \in W$
- $interpret(\varphi_1 \vee \varphi_2) = interpret(\varphi_1) \vee interpret(\varphi_2)$
- $interpret(\neg\varphi) = \neg(interpret(\varphi))$
- $interpret(\exists s.\varphi) = \exists s.(\psi_U(s) \wedge interpret(\varphi))$
- $interpret(\exists W.\varphi) = \exists W.((\forall s.(s \in W \Rightarrow \psi_U(s))) \wedge interpret(\varphi))$

It is not hard to show that for any formula δ on (L_v, L_e)-labeled graphs $Graph(Subtree(T, S))$ satisfies δ iff T satisfies $interpret(\delta)$.

Now, we give another transformation, that transforms an MSO formula δ to a formula $\delta'(X)$ over a free set-variable X such that for any tree $T = (V, \lambda)$ and any valid subset $S \subseteq V$, $Subtree(T, S)$ satisfies δ iff T satisfies $\delta'(X)$ when X is interpreted to be S. In other words, we can transform a formula that expresses a property of a subtree to a formula that expresses the same property on the subtree defined by the free variable X. The transformation is given by the following function $tailor$; the crucial transformation are the edge-formulas, which has to be interpreted as the edges of the subtree defined by X.

- $tailor_X(p) = p$
- $tailor_X(Q_a(s)) = Q_a(s)$, for every $a \in L_v$
- $tailor_X(E_i(s,t)) = \exists s' \ [E_i(s,s') \wedge s' \leq t \wedge \forall t'. ((t' \in X \wedge s' \leq t') \Rightarrow t \leq t')]$, for every $i \in [k]$.
- $tailor_X(s = t) = (s = t)$
- $tailor_X(s \in W) = s \in W$
- $tailor_X(\varphi_1 \vee \varphi_2) = tailor(\varphi_1) \vee tailor(\varphi_2)$
- $tailor_X(\neg\varphi) = \neg(tailor(\varphi))$
- $tailor_X(\exists s.\varphi) = \exists s.(s \in X \wedge tailor(\varphi))$
- $tailor_X(\exists W.\varphi) = \exists W.(W \subseteq X \wedge tailor(\varphi))$

The above transformation satisfies the following property. For any MSO sentence δ on k-ary trees, for any tree $T = (V, \lambda)$ and for any valid subset $S \subseteq V$, $Subtree(T, S)$ satisfies δ iff T satisfies $tailor_X(\delta)$ when X is interpreted to be S.

Note that the above transformations can be combined. For any MSO formula δ on (L_v, L_e) labeled graphs, consider the formula $tailor_X(interpret(\delta))$. Then for any tree $T = (V, \lambda)$ and for any valid subset $S \subseteq V$, $Graph(Subtree(T, S))$ satisfies δ iff T satisfies $tailor_X(interpret(\delta))$, where X is interpreted as S.

Expressing minimal models in MSO First, we can also express, with an MSO formula $ValidSubModel(X)$, with a free set variable X, that holds in a tree $T = (V, \lambda)$ iff X is interpreted as a valid submodel of T. This is easy; we express the properties of X being least-common ancestor closed, and also check that the subtree defined by X satisfies ψ_{Tr}:

$$ValidSubModel(X) =$$

$$\forall s, t, u\,((s \in X \wedge t \in X \wedge lca(s,t,u)) \Rightarrow u \in X) \wedge tailor_X(\psi_{Tr}) \\ \wedge (\forall s(s \in X \wedge tailor_X(\psi_U(s))) \;\Rightarrow\; \psi_U(s))$$

where $lca(s, t, u)$ is an MSO formula that checks whether u is the least-common ancestor of s and t in the tree; this expresses the three requirements for X to be a valid subset of nodes of a tree.

We are now ready to define the MSO formula on k-ary trees $MinModel$ that captures minimal models.

$$MinModel = \neg\exists X.(\,ValidSubModel(X) \wedge \\ \exists s.(s \in X) \;\wedge\; \exists s.(s \notin X) \;\wedge \\ (\,\forall\vec{y}\,\forall\vec{p}\,((\wedge_{y\in\vec{y}}(y \in S \wedge \varphi_U(y)) \wedge interpret(\widehat{\varphi}(\vec{y},\vec{p})) \\ \Rightarrow tailor_X(\widehat{\varphi}(\vec{y},\vec{p})))))$$

The above formula when interpreted on a tree T says that there does not exists a set X that defines a non-empty valid strict subset of the nodes of T, which defines a model $Graph(Subtree(T, S))$. Further, for every valuation of $\vec{y}$ over the nodes of $Graph(Subtree(T, S))$ and for every valuation of the Boolean variables $\vec{p}$ such that the structural abstraction of φ holds in $Graph(T)$, it demands that the same valuation also makes the structural abstraction of φ hold in $Graph(Subtree(T, S))$.

Note that the above is a *pure* MSO formula on trees, and encodes the properties required of a minimal model with respect to satisfiability-preserving embeddings. Using the classical logic-automaton connection [4], we can transform the MSO formula to a tree automaton that accepts precisely those trees that satisfy the formula. Since the finite-ness of the language accepted by a tree automaton is decidable, we can check whether the universal STRAND formula $\forall\vec{y}\;\varphi(\vec{y})$ has only a finite number of minimal models wrt satisfiability-preserving embeddings, and hence decide membership in the decidable fragment STRAND_{dec}.

THEOREM 5.6. *Given a sentence $\exists\vec{x}\forall\vec{y}\;\varphi(\vec{x},\vec{y})$, the problem of checking whether the sentence belongs to the fragment STRAND_{dec} is decidable.*

We in fact develop, using the tool MONA, the decision procedure above (see Section 7).

Deciding formulas in STRAND_{dec}:

We now give the decision procedure for satisfiability of sentences in STRAND_{dec} over a recursively defined class of data-structures. First, we transform the satisfiability problem to that of satisfiability of universal formulas of the form $\psi = \forall\vec{y}\;\varphi(\vec{y})$. Then, using the formula $MinModel$ described above, and by transforming it to tree automata, we extract the set of *all* trees accepted by the tree-automaton in order to get the tree-representation of all the minimal models. Note that this set of minimal models is finite, and the sentence is satisfiable iff it is satisfiable in some data-extension of one of these models.

We can now write a quantifier-free formula over the data-logic that asserts that one of the minimal model has a data-extension that satisfies ψ. This formula will be a disjunction of m sub-formulas $\eta_1, \ldots, \eta_m$, where m is the number of submodels. Each formula η_i will express that there is a data-extension of the i'th minimal model that satisfies ψ. First, since a minimal model has only a finite number of nodes, we create one data-variable for each of these nodes, and associate them with the nodes of the model. It is now not hard to transform the formula ψ to this model using no quantification. The universal quantification over $\vec{y}$ translates to a conjunction of formulas over all possible valuations of $\vec{y}$ over the nodes of the fixed model. Existential (universal) quantified variables are then "expanded" using disjunction (conjunction, respectively) of formulas for all possible valuations over the fixed model. The edge-relations between nodes in the model are interpreted on the tree using MSO formulas in $\mathcal{R}$, which are then expanded to conditions over the fixed set of nodes in the model. Finally, the data-constraints in the STRAND formula are directly written as constraints in the data-logic.

The resulting formula is a pure data-logic formula without quantification that is satisfiable if and only if ψ is satisfiable over $\mathcal{R}$. This is then decided using the decision procedure for the data-logic.

THEOREM 5.7. *Given a sentence $\exists\vec{x}\forall\vec{y}\,\varphi(\vec{x},\vec{y})$ over $\mathcal{R}$ in STRAND_{dec}, the problem of checking whether ψ is satisfiable reduces to the satisfiability of a quantifier-free formula in the data-logic. Since the quantifier-free data-logic is decidable, the satisfiability of STRAND_{dec} formulas is decidable.*

5.6 Syntactically-definable decidable logics

The fragment STRAND_{dec} that we proved decidable above is a semantically defined fragment, though membership in the fragment is effectively checkable. The problem in identifying decidable syntactic fragments of STRAND is that it is very hard to identify syntactically when the number of minimal models with respect to satisfiability-preserving embeddings will be finite, because of the myriad possibilities and complexities in the underlying class of recursive data-structures $\mathcal{R}$.

However, on certain classes of recursive structures we can show that *all* formulas in STRAND of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$, where φ does not have further quantification, is *decidable*. Let us call a STRAND formula *inner-quantification-free* if it is of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$, where φ is quantification-free.

Consider a class of recursive data-structures $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a\in L_v}, \{\beta_b\}_{b\in L_e})$, where $\psi_{Tr} = \texttt{true}$ (i.e. all nodes of the tree belong to the model). Now, assume that for *every* tree satisfying ψ_{Tr}, and for *every submodel* S of T, it turned out that for any two nodes $u, u' \in Graph(S)$, any edge $\beta_b(u, v)$ holds in $Graph(S)$ iff $\beta_b(u, v)$ holds in $Graph(T)$. Then notice that for any formula $\varphi(\vec{y})$ without quantification, for any valuation of $\vec{y}$ over S, the structural abstraction $\hat{\varphi}$ will hold in $Graph(S)$ iff it holds in $Graph(T)$. Hence S will necessarily structurally embed in T. Furthermore, it is not hard to see that the number of minimal models will be finite in this case (intuitively, the *smallest* trees accepted by the tree-automaton for ψ, where a smallest tree is defined as one where in no path a state repeats, will form the set of minimal models). Consequently, all formulas of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x},\vec{y})$, where φ has no quantification, will be decidable in $\mathcal{R}$.

For example, if $\mathcal{R}$ is a set of trees, and the signature had only relations $LeftSubtree(x, y)$ and $RightSubtree(x, y)$, then notice that in any subtree of a tree T, two nodes will be related by the $LeftSubtree$ relation iff they are related by the $LeftSubtree$ relation on the tree T (similarly for $RightSubtree$). Hence the syntactically defined class of inner-quantification-free STRAND formulas is decidable. Suppose in addition we had a list connecting the leaves of the tree, with next-pointer relation f, and the signature of $\mathcal{R}$ had only the reachability relation f^* on f, again the inner-quantification-free fragment of STRAND is decidable.

In fact, we can extend the above arguments further. Given *any* recursive structure $\mathcal{R}$, we can *decide* if $\mathcal{R}$ satisfies the above condition mentioned, using an appropriate formulas over MSO. Then, if $\mathcal{R}$ verifies against this check, then the syntactic fragment of inner-quantification-free fragment of STRAND is decidable on this class!

6. Program Verification Using STRAND

In this section we show how STRAND can be used to reason about the correctness of programs, in terms of verifying Hoare-triples where the pre- and post-conditions express both the structure of the heap as well as the data contained in them. The pre- and post-conditions that we allow are STRAND formulas that consist of Boolean combinations of the formulas with pure existential or pure universal quantification over the data-variables (i.e. Boolean combinations of formulas of the form $\exists \vec{x} \varphi$ and $\forall \vec{y} \varphi$); let us call this fragment $\text{STRAND}_{\exists,\forall}$.

Given a straight-line program P that does destructive pointer-updates and data updates, we model a Hoare triple as a tuple $(\mathcal{R}, Pre, P, Post)$, where the pre-condition is given by the data-structure constraint $\mathcal{R}$ with the $\text{STRAND}_{\exists,\forall}$ formula Pre, and the post-condition is given by the $\text{STRAND}_{\exists,\forall}$ formula $Post$ (note that structural constraints on the data-structure for the post-condition are also expressed in $Post$, using MSO logic).

In this section, we show that given such a Hoare triple, we can reduce checking whether the Hoare triple is *not* valid can be reduced to a satisfiability problem of a STRAND formula over a class of recursively defined data-structures $\mathcal{R}_P$. This then allows us to use $\text{STRAND}_{\exists,\forall}$ to verify programs (where, of course, loop-invariants are given by the programmer, which breaks down verification of a program to verification of straight-line code). Intuitively, this reduction *augments* the structures in $\mathcal{R}$ with extra nodes that could be created during the execution of P, and models the *trail* the program takes by logically defining the configuration of the program at each time instant. Over this trail, we then express that the pre-condition holds and the post-condition fails to hold. We also construct formulas that check if there is any memory access violation during the run of P (e.g. free-ing locations twice, dereferencing a null pointer, etc.).

Syntax of programs. Let us define the syntax of a basic programming language manipulating heaps and data; more complex constructs can be defined by combining these statements appropriately. Let Var be a countable set of *pointer variables*, F be a countable set of *structural pointer fields*, and `data` be a *data field*. A *condition* is defined as follows: (for technical reasons, negations are pushed all the way in):

$$\psi \in \texttt{Cond} ::= \gamma(q^1.\texttt{data}, \ldots, q^k.\texttt{data}) \mid \neg\gamma(q^1.\texttt{data}, \ldots, q^k.\texttt{data}) \\ \mid\ p == q \mid p \neq q \mid p == nil \mid p \neq nil \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2$$

where $p, q, q^1, \ldots, q^k \in Var$, and γ is a predicate over data values. The set of statements `Stmt` defined over Var, F, and `data` is defined as follows:

$$s \in \texttt{Stmt} ::=\ p := new \mid free(p) \mid assume(\psi) \mid p := nil \mid \\ p := q \mid p.f := q \mid p := q.f \mid p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$$

where $p, q, q^1, \ldots, q^k \in Var$, $f \in F$, h is a function over data, and ψ is a condition. A *program* P over Var, F, and `data` is a non empty finite sequence of statements $s_1; s_2; \ldots; s_m$, with $s_i \in \texttt{Stmt}$.

The semantics of a program is the natural one and we skip its definition.

Let $\mathcal{R}$ be a recursive data-structure, $Pre, Post$ be two $\text{STRAND}_{\exists,\forall}$ formulas, and $P ::= s_1; s_2; \ldots; s_m$ be a program. The configuration of the program at any point is given by a heap modeled as a graph, where nodes of the graph are assigned data values. For a program with m statements, let us fix the configurations to be $G_0, \ldots, G_m$.

The trail. The idea is to capture the entire computation starting from a particular data-structure using a single data-structure. The main intuition is that if we run P over a graph $G_0 \in Graph(\mathcal{R})$ then a new class of recursive data-structures $\mathcal{R}_P$ will define a graph G_{trail} which encodes in it G_0, as well as all the graphs G_i, for every $i \in [m]$. G_{trail} has the same nodes of G plus m other fresh nodes (these nodes will be used to model newly created nodes P creates as well as to hold new data-values of variables that are assigned to in P). Each of these new nodes are pointed by a distinguished pointer variable new_i. Initially, these additional nodes are all inactive in G_0. We build an MSO-defined unary predicate $active_i$ that captures at each step i the precise set of active nodes in the heap. To capture the pointer variables at each step of the execution, we define a new unary predicate p_i, for each $p \in Var$ and $i \in [0, m]$. Similarly, we create MSO-defined binary predicates f_i for each $f \in F$ and $i \in [0, m]$, to capture structural pointer fields at step i. The heap G_i at step i is hence the graph consisting of all the nodes x of G_{trail} such that $active_i(x)$ holds true, and the pointers and edges of G_i are defined by p_i and f_i predicates, respectively.

Formally, fix a recursively defined data-structure $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_p\}_{p \in Var}, \{\beta_f\}_{f \in F})$, with a monadic predicate α_{xnil}, which evaluates to a unique NIL node in the data-structure. Then its trail with respect to the program P is defined as $\mathcal{R}_P = (\psi'_{Tr}, \psi'_U, \{\alpha'_p\}_{p \in Var'}, \{\beta'_f\}_{f \in F'})$ where:

- ψ'_{Tr} is designed to hold on all trees in which the first subtree of the root satisfies ψ_{Tr} and the second child of the root has a chain of $m-1$ nodes where each of them is the second child of the parent.
- ψ'_U holds true on the root, on all the second child descendent of the root, and on all first child descendent on which ψ_U holds true.
- $Var' = \{new_i | i \in [m]\} \cup \{p_i | p \in Var, i \in [0, m]\}$, and
  -(1) α'_{new_1} holds only on the root, and α'_{new_i} holds true only on the $i+1$'th descendent of the second child of the root, for every $i \in [m-1]$.
  -(2) for every $p \in Var$ and $i \in [m]$, $\alpha'_{p_0} = \alpha_p$ and α'_{p_i} is defined as in **Figure 2**.
- $F' = \{f_i | f \in F, i \in [0, m]\}$, and for every $f \in F$ and $i \in [m]$, $\beta'_{f_0} = \beta_f$ and β'_{f_i} is defined as in **Figure 2**.

In **Figure 2**, the MSO formulas α'_{p_i} and β'_{f_i} are derived in the natural way from the semantics of the statements, except for the statement $p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$. Although the semantics for this statement does not involve any structural modification of the graph (it changes only the data value associated p), we represent this operation by making a *new version* of the node pointed by p in order to represent explicitly the change for the data value corresponding to that node. We deactivate the node pointed by p_{i-1} and activate the dormant node pointed by new_i. All the edges in the graph and the pointers are rearranged to reflect this exchange of nodes.

In **Figure 2**, we also define two more MSO formulas, $active_i$ and $error_i$, which are not part of the trail, where the first models the active nodes at step i, and the second expresses when an error occurs due to the dereferencing of a variable pointing to $xnil$, respectively.

Handling data constraints. The trail $\mathcal{R}_P$ captures all the structural modifications made to the graph during the execution P. However, data constrains entailed by $assume$ statements and data-assignments cannot be expressed in the trail as they are not express-

Program	Verification condition	Structural solving (MONA)					Data-constraint Solving (Z3 with QF-LIA)			
		in STRAND$_{dec}$? (finitely-many minimal models)	#States	Final BDD size	Time(s)	Graph model exists?	Bound (#Nodes)	Formula size (KB)	Satisfiable?	Time(s)
sorted-list-search	before-loop	Yes	67	264	0.34	No	-	-	-	-
	in-loop	Yes	131	585	0.59	No	-	-	-	-
	after-loop	Yes	67	264	0.18	No	-	-	-	-
sorted-list-insert	before-head	Yes	73	298	1.66	Yes	5	6.2	No	0.02
	before-loop	Yes	259	1290	0.38	No	-	-	-	-
	in-loop	Yes	1027	6156	4.46	No	-	-	-	-
	after-loop	Yes	146	680	13.93	Yes	7	14.5	No	0.02
sorted-list-insert-error	before-loop	Yes	298	1519	0.34	Yes	7	9.5	Yes	0.02
sorted-list-reverse	before-loop	Yes	35	119	0.24	No	-	-	-	-
	in-loop	Yes	513	2816	2.79	No	-	-	-	-
	after-loop	Yes	129	576	0.35	No	-	-	-	-
bst-search	before-loop	Yes	52	276	5.03	No	-	-	-	-
	in-loop	Yes	160	1132	32.80	Yes	9	7.7	No	0.02
	after-loop	Yes	52	276	3.27	No	-	-	-	-
bst-insert	before-loop	Yes	36	196	1.34	No	-	-	-	-
	in-loop	Yes	68	452	9.84	No	-	-	-	-
	after-loop	Yes	20	84	1.76	No	-	-	-	-

Figure 3. Results of program verification

ible in MSO. We impose them in the STRAND formula. We define a formula φ_i for each statement index $i \in [m]$, where if s_i is not an assume or a data-assignment statement, then $\varphi_i = \texttt{true}$. Otherwise, there are two cases:

Handling assume assignments. If s_i is the statement $assume(\psi)$, then φ_i is the STRAND formula obtained by *adapting* the constraint φ to the i'th stage of the trail. This is not hard, but is tedious, and we skip its definition. Constraints on data-variables asserted in the formula using data-logic constraints.

Handling data-assignments. The STRAND formula φ_i for a data-assignment statement $p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$ is:

$$\varphi_i \quad := \quad \exists ex, ex_1, \ldots, ex_k.\ p_i(ex) \wedge$$

$$(\bigwedge_{i \in [k]} q^j_{i-1}(ex_j)) \wedge \texttt{data}(ex) = h(\texttt{data}(ex_1), \ldots, \texttt{data}(ex_k))$$

which translates s_i into STRAND making sure that it refers to the heap at step i–1.

Adapting pre and post conditions to the trail. The last ingredient that we need is to express the STRAND$_{\exists,\forall}$ formulas Pre and the negation of the $Post$ on the trail $\mathcal{R}_P$. More specifically, we need to adapt Pre to the trail for index 0, which corresponds to the original graph, i.e. the predicates p are replaced with p_0, for every $p \in Var$, and the edge predicates f with f_0, for every $f \in F$. Moreover, whenever we refer to a node in the graph we need to be sure that node is active which can be done by using the predicate $active_0(x)$ which holds true if x is in the first subtree of the root and $\psi'_U(x)$ holds. A similar transformation is done for the formula $\neg Post$, where now we consider pointers, edge labels, and active nodes at the last step m. Let $Pre_{\mathcal{R}_P}$ (resp., $Post_{\mathcal{R}_P}$) be the STRAND formula corresponding to the adaptation of Pre (resp., $Post$)

Reduction to satisfiability problem on the trail. It is easy to see that an error occurs during the execution of P on a graph defined through $\mathcal{R}$ that satisfies Pre if the following STRAND formula is satisfiable on the trail $\mathcal{R}_P$:

$$Error = \bigvee_{i \in [m]} \left(Pre_{\mathcal{R}_P} \wedge \bigwedge_{j \in [i-1]} \varphi_j \quad \wedge \quad error_i \right)$$

Similarly, the Hoare triple is not valid iff the following STRAND formula is satisfiable on the trail:

$$Violate_{Post} = Pre_{\mathcal{R}_P} \wedge (\bigwedge_{j \in [m]} \varphi_j) \wedge \neg Post_{\mathcal{R}_P}$$

THEOREM 6.1. *Let P be a program, $\mathcal{R}$ be a recursive data-structure, and $Pre, Post$ be two* STRAND *formulas over Var, F, and* data*. Then, there is a graph $G \in Graph(\mathcal{R})$ that satisfies Pre and where either P terminates with an error or the obtained graph G' does not satisfy $Post$ iff the* STRAND *formula $Error \vee Violate_{Post}$ is satisfiable on the trail $\mathcal{R}_P$.*

7. Evaluation

7.1 Implementation

In this section, we demonstrate the effectiveness and practicality of the decision procedure for STRAND$_{dec}$ by checking verification conditions generated in proving properties of several heap-manipulating programs. Given pre-conditions, post-conditions and loop-invariants, each linear block of statements of a program yields a Hoare triple, which is manually translated into a universal STRAND formula ψ over trees and integer arithmetic, as a verification condition. The decision procedures consists of a structural phase, where we decide if the formula is in the decidable fragment, and if it is, build an automaton representing all finite models. This phase is effected by using MONA [9], a monadic second-order logic solver over (strings and) trees. In the second data-constraint solving phase, the finite models, if any, are examined by the data-solver Z3 [17] to check if they can be extended with data-values to satisfy the formula.

Instead of building an automaton representing the minimal models and then checking it for finiteness, we check the finiteness formula $MinModel_\psi$ using WS2S, supported by MONA, which is a monadic second-order logic over *infinite* trees with set-quantification restricted to finite sets. By quantifying over a finite universe U, and transforming all quantifications to be interpreted over U, we can interpret $MinModel_\psi$ over all finite trees. Let us denote this emulation as $MinModel'_{U,\psi}$. The finiteness condition can now be checked by asking if *there exists a finite set B* such that any minimal model for ψ is contained within the nodes of B:

$$\exists\ Bound\ \forall U\ \ \forall Q_{a_{(a \in \Sigma)}}\ (MinModel'_{U,\psi} \Rightarrow (U \subseteq Bound))$$

$[p := new\ ]$:

$$\alpha'_{p_i}(x) = \alpha'_{new_i}(x), \quad \alpha'_{q_i}(x) = \alpha'_{q_{i-1}}(x), \ \forall q \in Var \setminus \{p\},$$
$$\beta'_{f_i}(x,y) = \beta'_{f_{i-1}}(x,y), \ \ active_i(x) = active_{i-1}(x) \vee \alpha'_{new_i}(x)$$
$$error_i = \texttt{false}$$

$[free(p)\ ]$:

$$\alpha'_{z_i}(x) = (\alpha'_{z_{i-1}}(x) \wedge (\alpha'_{xnil_{i-1}}(x) \vee \neg\alpha'_{p_{i-1}}(x))) \vee (\alpha'_{xnil_{i-1}}(x) \wedge \neg\alpha'_{z_{i-1}}(x))$$
$$\beta'_{f_i}(x,y) = (\beta'_{f_{i-1}}(x,y) \wedge \neg\alpha'_{p_{i-1}}(x)) \vee (\alpha'_{xnil_{i-1}}(y) \wedge \exists ex.\ (\beta'_{f_{i-1}}(x,ex) \wedge \alpha'_{p_{i-1}}(ex)))$$
$$active_i(x) = active_{i-1}(x) \wedge \neg\alpha'_{p_{i-1}}(x)$$
$$error_i = \forall x.\ \alpha'_{p_{i-1}}(x) \Rightarrow \neg\alpha'_{xnil_{i-1}}(x)$$

$[p := nil\ ]$:

$$\alpha'_{p_i(x)} = \alpha'_{xnil_{i-1}}(x), \ \ \alpha'_{z_i}(x) = \alpha'_{z_{i-1}}(x), \ \ \forall z \in Var \setminus \{p\}$$
$$\beta'_{f_i}(x,y) = \beta'_{f_{i-1}}(x,y), \ \ active_i(x) = active_{i-1}(x), \ \ error_i = \texttt{false}$$

$[p := q\ ]$:

$$\alpha'_{p_i}(x) = \alpha'_{q_i}(x), \ \ \alpha'_{z_i}(x) = \alpha'_{z_{i-1}}(x), \ \ \forall z \in (Var \setminus \{p\})$$
$$\beta'_{f_i}(x,y) = \beta'_{f_{i-1}}(x,y), \ \ active_i(x) = active_{i-1}(x), \ \ error_i = \texttt{false}$$

$[p.f := q\ ]$:

$$\alpha'_{z_i}(x) = \alpha'_{z_{i-1}}(x), \qquad \forall z \in Var$$
$$\beta'_{f_i}(x,y) = (\neg\alpha'_{p_{i-1}}(x) \wedge \beta'_{f_{i-1}}(x,y)) \vee (\alpha'_{p_{i-1}}(x) \wedge \alpha'_{q_{i-1}}(y))$$
$$\alpha'_{g_i}(x,y) = \alpha'_{g_{i-1}}(x,y), \qquad \forall g \in (F \setminus \{f\})$$
$$active_i(x) = active_{i-1}(x), \ \ error_i = \exists x.(\alpha'_{p_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x))$$

$[p := q.f\ ]$:

$$\alpha'_{p_i(x)} = \exists ex.\ (\alpha'_{q_{i-1}}(ex) \wedge \beta'_{f_{i-1}}(ex,x))$$
$$\alpha'_{q_i}(x) = \alpha'_{q_{i-1}}(x), \qquad \forall q \in (Var \setminus \{p\})$$
$$\beta'_{f_i}(x,y) = \beta'_{f_{i-1}}(x,y)$$
$$\alpha'_{g_i}(x,y) = \alpha'_{g_{i-1}}(x,y), \qquad \forall g \in (F \setminus \{f\})$$
$$active_i(x) = active_{i-1}(x), \ \ error_i = \exists x.\ (\alpha'_{q_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x))$$

$[assume(\psi)\ ]$:

$$\alpha'_{q_i}(x) = \alpha'_{q_{i-1}}(x), \forall q \in Var, \quad \beta'_{f_i}(x,y) = \beta'_{f_{i-1}}(x,y), \forall f \in F$$
$$active_i(x) = active_{i-1}(x), \ \ error_i = \exists x. \bigvee_{p \in Var^{\psi}} (\alpha'_{p_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x))$$

where Var^{ψ} is the set of all variables occurring in ψ.

$p.\texttt{data} := h(q^1.\texttt{data}, \ldots, q^k.\texttt{data})$:

$$\alpha'_{p_i}(x) = \alpha'_{new_i}(x), \quad \alpha'_{q_i}(x) = \alpha'_{q_{i-1}}(x), \ \ \forall q \in Var \setminus \{p\}$$
$$\beta'_{f_i}(x,y) = (\beta'_{f_{i-1}}(x,y) \wedge \neg\alpha'_{p_{i-1}}(x)) \vee (\alpha'_{new_i}(y) \wedge \exists ex.\ (\beta'_{f_{i-1}}(x,ex) \wedge \alpha'_{p_{i-1}}(ex)))$$
$$active_i(x) = (active_{i-1}(x) \wedge \neg p_{i-1}(x)) \vee \alpha'_{new_i}(x)$$
$$error_i = \exists x.(\bigvee_{z \in \{p, q^1, \ldots, q^k\}} (\alpha'_{z_{i-1}}(x) \wedge \alpha'_{xnil_{i-1}}(x)) \ )$$

Figure 2. Predicates defining the new data-structure.

This formula has no free-variables, and hence either holds on the infinite tree or not, and can be checked by MONA. This formula evaluates to `true` iff the formula is in STRAND$_{dec}$.

We also follow a slightly different procedure to synthesize the data-logic formula. Instead of extracting each finite model, and checking if there is a data-extension for it, we obtain a *bound* on the model, and ask the data-solver to check for any model within that bound. This is often a much simpler formula to feed to the data-solver.

If finiteness condition above evaluates to `true`, then we first check the satisfiability of the structural abstraction $\widehat{\psi}$ of ψ. If it is unsatisfiable, then by Proposition 5.2, ψ is also unsatisfiable. Otherwise, we ask MONA for the minimal $Bound$ of minimal models, which will be a prefix-closed set of nodes of the tree. Given $Bound$, ψ is manually translated into a quantifier free formula over integer arithmetic, which is fed to Z3.

7.2 Experiments

Figure 3 presents the evaluation of our tools on checking a set of programs that manipulate either sorted singly-linked lists or binary search trees. Note that the binary search trees presented here are out of the scope of the logics HAVOC [12] and CSL [5].

The programs `sorted-list-search` and `sorted-list-insert` search and insert a node in a sorted singly-linked list, respectively, while `sorted-list-insert-error` is the insertion program with an intended error. The program `sorted-list-reverse` is a routine for in-place reversal of a sorted singly-linked list, which results in a reverse-sorted list. The routines `bst-search` and `bst-insert` search and insert a node in a binary search tree, respectively.

For all these examples, a set of partial correctness properties including both structural and data requirements is checked. For example, assuming a node with value k exists, we check if both `sorted-list-search` and `bst-search` return a node with value k. For `sorted-list-insert`, we assume that the inserted value does not exist, and check if the resulting list contains the inserted node, and the sortedness property continues to hold. In the program `bst-insert`, assuming the tree does not contain the inserted node in the beginning, we check whether the final tree contains the inserted node, and the binary-search-tree property continues to hold. In `sorted-list-reverse`, we check if the output list is a valid list that is reverse-sorted.

Note that each program requires checking several verification conditions (usually for the linear block from the beginning of the program to a loop, the loop invariant linear block, as well as the loop invariant to the end of the program).

The experiments were conducted on a 2.2GHz, 4GB machine running Windows 7. For the structural solving phase, we report first whether the verification condition falls within our decidable fragment STRAND$_{dec}$. We also report the number of states, the BDD sizes to represent automata, and the time taken by MONA to compute the minimal models. We report whether there were *any* models found; note that if the formula is unsatisfiable and there are no models, the Z3 phase can be skipped (these are denoted by "-" annotations in the table for Z3).

For the data-constraint solving phase, we first report the number of nodes of the tree (or string) which forms an upper bound for all minimal models. The Z3 formulas are typically large (but simple) as one can see from the size of the formulas in the table. We report whether Z3 found the formula to be satisfiable or not ((all cases were unsatisfiable, except `sorted-list-insert-error` as the Hoare-triples verified were correct)), and the time it took to determine this.

In the `sorted-list-insert-error` program, we removed the initial code that checks whether the inserted value is greater than the value at the head of the list, and inserts the element before the head. The Z3 phase failed and gave as a counter-example a two-element list, with value at head 0 and the value of the inserted value is -6. The `bst-insert` routine verifies without going to the Z3 phase, which means that the insertion correctness of the routine is *independent* of the data-solver entirely (not even the transitivity of $\leq$ on integers is needed). The search routine however does require help from the Z3 solver.

The experimental results show that natural verification conditions tend to in the decidable fragment STRAND$_{dec}$. Moreover, the expressiveness of our logic allows us to express complex conditions involving structure and data, and yet are handled well by MONA and Z3. We believe that a full-fledged engineering of an SMT solver for STRAND$_{dec}$, that works as an independent SMT solver for answering queries involving heap structures and data, is a promising future direction encouraged by the above results.

8. Related Work

We first discuss related work that can reason combinations of heaps and data. In handling heaps, first order theories that can reason with restricted forms of the reachability relation for ensuring decidability are the most common. The work most closely related to our work is the logic in HAVOC, called LISBQ [12], that offers a reasoning with generic heaps combined with an arbitrary data-logic. The logic has restricted reachability predicates and universal quantification, but is syntactically severely curtailed, to obtain decidability. We find the restrictions on the syntax quite awkward, with *sort*-based restrictions in the logic. Furthermore, the logic cannot handle even simple constraints over trees with unbounded depth where the nodes are of the same sort(like a tree being a binary search-tree). However, the logic is extremely efficient, as it uses no structural solver, but translates the structure-solving also to the (Boolean aspect) of the SMT solver. We gained a lot of insight into decidability by studying the expressive power of HAVOC, and we believe that STRAND generalizes some of the underlying ideas present in HAVOC to a much more powerful technique for decidability. The logic CSL [5] has a similar flavor as HAVOC, with similar sort-restrictions on the syntax, but generalizes to handle doubly linked lists, and allows size constraints on structures.

Rakamarić et al [20] propose an inference rule system for reasoning with restricted reachability (but this logic does not have universal quantification and cannot express disjointness constraints), and an SMT solver based implementation has been reported [21]. Restricted forms of reachability were first axiomatized in early work by Nelson [18]. Several mechanisms without quantification exist, including the work reported in [22, 1]. Automatic decision procedures that approximate higher-order logic using first-order logic, using approximate logics over sets and their cardinalities, have been proposed [11].

There is a rich literature on heap analysis without data. Since first-order logic over graphs is undecidable, decidable logics must either restrict the logic or the class of graphs. The closest work to ours in this realm is PALE [16], which restricts structures to be definable over tree-skeletons, similar to STRAND, but support much more expressive monadic second-order constraints (but not with data), using the MONA system [9]. PALE can prove several heap-manipulating examples correct, but by manually abstracting the data domain. Several approximations of first-order axiomatizations of reachability have been proposed: axioms capturing *local* properties [15], a logic on regular patterns that is decidable [25], among others.

Finally, separation logic [23] has emerged as a convenient logic to express heap properties of programs, and a decidable fragment (without data) on lists is known [3]. However, not many extensions of separation logics handle data constraints (see [14] which combines this logic for linked lists with arithmetic). We hope to extend the technique presented here to study decidable fragments of separation logic combined with data constraints in the future.

References

[1] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, volume 3385 of *LNCS*, pages 164–180, 2005.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01*, pages 203–213, New York, NY, USA, 2001. ACM.

[3] J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. *In FSTTCS*, pages 97–109, 2004.

[4] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 2001.

[5] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR*, Vol. 5710 *LNCS*, pages 178–195, 2009.

[6] J. Engelfreit. Context-Free Graph Grammars *In Handbook of Formal Languages, Vol. 3, Chapter 3.*. Springer, 1997.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02*, pages 234–245, New York, NY, USA, 2002. ACM.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05:*, pages 213–223, 2005. ACM.

[9] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Available from `http://www.brics.dk/mona/`.

[10] N. Klarlund and M. I. Schwartzbach. Graph types. In *POPL '93*, pages 196–205, New York, NY, USA, 1993. ACM.

[11] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, 2007.

[12] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08*, pages 171–182, New York, NY, USA, 2008. ACM.

[13] T. Lev-Ami, and M. Sagiv. Tvla: A system for implementing static analyses. *IN STATIC ANALYSIS SYMP*, pages 280–301, 2000.

[14] S. Magill, M. Tsai, P. Lee, and Y. Tsay. Thor: A tool for reasoning about shape and arithmetic. *IN CAV, LNCS 5123*, pages 428–432, 2008.

[15] S. Mcpeak, and G. C. Necula. Data structure specifications via local equality axioms. *IN CAV*, pages 476–490, 2005.

[16] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01*, pages 221–231, New York, NY, USA, 2001. ACM.

[17] L. De Moura, and N. Bjørner. Z3: An efficient smt solver. *TACAS*, 2008.

[18] G. Nelson. Verifying reachability invariants of linked structures. In *POPL '83*, pages 38–47, New York, NY, USA, 1983. ACM.

[19] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[20] Z. Rakamaric, J. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI'07*, pages 106–121, 2007. Springer-Verlag.

[21] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an smt framework. In *ATVA'07*, pages 237–252, 2007. Springer-Verlag.

[22] S. Ranise and C. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 206 –215, 11-15 2006.

[23] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS, 2002.*, pages 55 – 74, 2002.

[24] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. *IN 10TH TACAS*, pages 530–545, 2004.

[25] G. Yorsh, A. M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. In *FoSSaCS*, Vol. 3921 of *LNCS*, pages 94–110, 2006.