

A Decidable Logic for Tree Data-Structures with Measurements

Xiaokang Qiu and Yanjun Wang

Purdue University
{xkqiu, wang3204}@purdue.edu

Abstract. We present DRYAD_{dec} , a decidable logic that allows reasoning about tree data-structures with measurements. This logic supports user-defined recursive measure functions based on Max or Sum, and recursive predicates based on these measure functions, such as AVL trees or red-black trees. We prove that the logic’s satisfiability is decidable. The crux of the decidability proof is a small model property which allows us to reduce the satisfiability of DRYAD_{dec} to quantifier-free linear arithmetic theory which can be solved efficiently using SMT solvers. We also show that DRYAD_{dec} can encode a variety of verification and synthesis problems, including natural proof verification conditions for functional correctness of recursive tree-manipulating programs, legality conditions for fusing tree traversals, synthesis conditions for conditional linear-integer arithmetic functions. We developed the decision procedure and successfully solved 220+ DRYAD_{dec} formulae raised from these application scenarios, including verifying functional correctness of programs manipulating AVL trees, red-black trees and treaps, checking the fusibility of height-based mutually recursive tree traversals, and counterexample-guided synthesis from linear integer arithmetic specifications. To our knowledge, DRYAD_{dec} is the first decidable logic that can solve such a wide variety of problems requiring flexible combination of measure-related, data-related and shape-related properties for trees.

1 Introduction

Logical reasoning about tree data-structures has been needed in various application scenarios such as program verification [32,24,26,42,49,4,14], compiler optimization [17,18,9,44] and webpage layout engines [30,31]. One particular class of desirable properties is the measurements of trees such as the size or height. For example, one may want to check whether a compiler optimizer always reduces the size of the program in terms of the number of nodes in the AST, or a tree balancing routine does not increase the height of the tree. These measurements are usually tangled with other shape properties and arithmetic properties, making logical reasoning very difficult. For example, an AVL tree should be sorted (arithmetic property) and height-balanced (shape property based on height), or a red-black tree of height 5 should contain at least 10 nodes (two measurements combined).

Most existing logics for trees either give up the completeness, aiming at mostly automated reasoning systems [5,16,39,4], or disallow either data properties [32,58,28] or tree measurements [24,25]. There do exist some powerful automatic verification systems that are capable of handling all of data, shape and tree measurements, such as VCDryad [26,42,36] and Leon [49,50]. However, the underlying logic of VCDryad cannot reason about the properties of AVL trees or red-black trees in a decidable fashion. In other words, they can verify the functional correctness of programs manipulating AVL trees or red-black trees, but they do not guarantee to provide a concrete counterexample to disprove a defective program. Leon [49,50] does guarantee decidability/termination for a small and brittle fragment of their specification language, which does not capture even the simplest measurement properties. For example, consider a program that inserts a new node to the leftmost path of a full tree: Skipping lines 2 and 3, the program recursively finds the leftmost leaf of the input tree and inserts a newly created node to the left. The **requires** (line 2) and **ensures** (line 3) clauses describe the simplest properties regarding the size of the tree: if the input tree is a nonempty full tree, the returned tree after running the program should not be a full tree and should contain at least 2 nodes. Note that the full-treeness $full^*$ and the tree-size $size^*$ can be defined recursively in VCDryad or Leon in a similar manner. However, none of VCDryad or Leon can verify the program below in a decidable fashion (see explanation in Section 5).

```

1 loc insertToLeft(Node t)
2   requires  $full^*(t) \wedge size^*(t) \geq 1$ 
3   ensures  $\neg full^*(ret) \wedge size^*(ret) \geq 2$ 
4   {
5     if (t.l == nil) t.l = new Node();
6     else t.l = insertToLeft(t.l);
7     return t;
8   }
```

In this work, our aim is to develop a decidable logic for tree data-structures that combines shape, data, and measurement. The decidability for such a powerful logic is highly desirable, as the decision procedure will guarantee to construct either a proof or witness trees as a disproof, which can benefit a wide variety of techniques beyond deductive verification, e.g.,

syntax-guided synthesis or test generation.

The decidable logic we set forth in this paper stems from the DRYAD logic, an expressive tree logic proposed along with a proof methodology called Natural Proofs [26,42]. DRYAD allows the user to define recursive definitions that can be unfolded exhaustively for arbitrarily large trees. Natural proofs, as a lightweight, automatic but incomplete proof methodology, restricts the unfolding to the footprint of the program only, then encodes the unfolded formula to decidable SMT-solvable theories using predicate abstraction, i.e., treating the remaining recursive definitions as uninterpreted. The limited unfolding and predicate abstraction make the procedure incomplete.

In this paper, we identify $DRYAD_{dec}$, a fragment of DRYAD, and show that its satisfiability is decidable. The fragment limits both user-defined recursive definitions and formulae with carefully crafted restrictions to obtain the *small model property*. With a given $DRYAD_{dec}$ formula, one can analytically compute a bound up to which all recursive definitions should be unfolded, and the small

$$\begin{array}{llll}
dir \in Loc \text{ Fields} & G \in Loc \text{ Field Groups} & x, y \in Loc \text{ Variables} & K : Int \text{ Constant} \\
f \in Int \text{ Fields} & r : Intermittence & j, k \in Int \text{ Variables} & q \in Boolean \text{ Variables}
\end{array}$$

$$\begin{array}{l}
\text{Increasing } Int \text{ function : } mij^*(x) \stackrel{def}{=} \mathbf{ite} \left(\mathbf{isNil}(x), -\infty, \max(\{mij^*(x.dir) \mid dir \in Dir\} \cup \{it[x]\}) \right) \\
\text{Decreasing } Int \text{ function : } mdj^*(x) \stackrel{def}{=} \mathbf{ite} \left(\mathbf{isNil}(x), \infty, \min(\{mdj^*(x.dir) \mid dir \in Dir\} \cup \{it[x]\}) \right) \\
\text{Increasing } IntSet \text{ function : } sj^*(x) \stackrel{def}{=} \mathbf{ite} \left(\mathbf{isNil}(x), \emptyset, (\bigcup_{dir} sj^*(x.dir)) \cup ST[x] \right) \\
\text{Measure function Max-based : } lij^*(x) \stackrel{def}{=} \mathbf{ite} \left(\mathbf{isNil}(x), 0, \max_{dir \in Dir} lij^*(x.dir) + \mathbf{ite}^r(v[x], 1, 0) \right) \\
\text{Measure function Sum-based : } eij^*(x) \stackrel{def}{=} \mathbf{ite} \left(\mathbf{isNil}(x), 0, \sum_{dir \in Dir} eij^*(x.dir) + \mathbf{ite}^r(v[x], 1, 0) \right) \\
\text{General predicate : } gp^*(x) \stackrel{def}{=} \mathbf{ite} \left(\mathbf{isNil}(x), \mathbf{true}, \left(\bigwedge_{dir} gp^*(x.dir) \right) \wedge \varphi[x.dir, x.f] \right) \\
(\varphi \text{ may involve other general predicates or increasing functions that only have positive} \\
\text{coefficients, or decreasing functions that only have negative coefficients.}) \\
\text{Measure-related predicate : } mp^*(x) \stackrel{def}{=} \mathbf{ite} \left(\mathbf{isNil}(x), \mathbf{true}, \left(\bigwedge_{dir} mp^*(x.dir) \right) \wedge \varphi[x.dir, x.f] \right) \\
(\varphi \text{ may involve anything allowed for general predicates and one Max-based measure function} \\
lij^* \text{ in the form of } lij^*(x.dir_1) - lij^*(x.dir_2) \geq K)
\end{array}$$

$$\begin{array}{l}
\text{Local } Int \text{ Term: } it, it_1, it_2, \dots ::= K \mid x.f \mid t_1 + t_2 \mid -t \mid \mathbf{ite}(v, t_1, t_2) \\
\text{Local } Set \text{ Term: } ST, ST_1, ST_2, \dots ::= \emptyset \mid \{it\} \mid ST_1 \cup ST_2 \mid ST_1 \cap ST_2 \\
\text{Local Formula: } v, v_1, v_2, \dots ::= it_1 \geq 0 \mid v_1 \wedge v_2 \mid v_1 \vee v_2 \mid \neg v
\end{array}$$

Fig. 1: Templates of DRYAD_{dec} Functions and Predicates

model property ensures that a fixed number of unfolding is sufficient and guarantees completeness. The DRYAD_{dec} logic features the following properties: a) allows user-defined and mutually recursive definitions to describe the functional properties of AVL trees, red-black trees and treaps; b) the satisfiability problem is *decidable*; c) experiments show that the logic can be used to encode and solve a variety of practical problems, including *correctness verification, fusibility checking and syntax-guided synthesis*. To the best of our knowledge, DRYAD_{dec} is the first decidable logic that can reason about a flexible mixture of sophisticated data, shape and measure properties of trees.

2 A decidable fragment of DRYAD

DRYAD is a logic for reasoning about tree data-structures, first proposed by Madhusudan *et al.* [26]. DRYAD can be viewed as a variant of first-order logic extended with least fixed points. The syntax of DRYAD is free of quantifiers but supports user-provided recursive functions for describing properties and measurements of tree data structures. Each recursive function maps trees to a boolean value, an integer or a set of integers, and is defined recursively in the following form: $F^*(x) \stackrel{def}{=} \mathbf{ite}(\mathbf{isNil}(x), F_{base}, F_{ind})$, where F_{base} stands for the value of the base case, i.e., x is *nil*, and F_{ind} recursively defines the value of $F^*(x)$ based on the local data fields and subtrees of x . DRYAD is in general undecidable and Mad-

$$\begin{aligned}
\text{Int Term: } t, t_1, t_2, \dots &::= K \mid j \mid \text{mj}^*(x) \mid \text{mdj}^*(x) \mid t_1 + t_2 \mid -t \mid \text{ite}(l, t_1, t_2) \\
\text{IntSet Term: } S, S_1, S_2, \dots &::= \emptyset \mid \{t\} \mid \text{sf}^*(x) \mid S_1 \cup S_2 \mid S_1 \cap S_2 \\
\text{Measure-related Formula } \psi &::= \text{lij}^*(x) - \text{lij}^*(y) \geq K \mid \text{eij}^*(x) - \text{eij}^*(y) \geq K \mid \\
&\quad \text{lij}^*(y) \geq K \mid \text{eij}^*(y) \geq K \mid \text{mp}^*(x) \\
&\quad (x \text{ is related to } \text{lij}^*, \text{eij}^*, \text{ or } \text{mp}^*, \text{ respectively.}) \\
\text{Negatable Formula: } l &::= q \mid t \geq 0 \mid t \in S \mid \psi \mid \text{isNil}(x) \mid \text{gp}^*(x) \mid \neg l \\
\text{Formula: } \varphi, \varphi_1, \varphi_2, \dots &::= l \mid S_1 \not\subseteq S_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2
\end{aligned}$$

(Every variable x can be related to only one measure function.)

Fig. 2: Syntax of DRYAD_{dec} logic

husudan *et al.* [26] present an automatic but incomplete procedure for DRYAD based on a methodology called *Natural Proofs*.

In this paper, we carefully crafted a decidable fragment of DRYAD, called DRYAD_{dec}, which is amenable for reasoning about the measurement of trees.

2.1 Syntax

The templates for recursive functions and predicates allowed in DRYAD_{dec} are shown in Figure 1 and the syntax of DRYAD_{dec} is presented in Figure 2. To simplify the presentation, these figures show unary functions and predicates only, i.e., those recursively defined over a single tree. DRYAD_{dec} also supports recursive functions and predicates with multiple arguments, which are amenable to define data structures characterizing loop invariants, such as list segments, tree-with-a-hole, etc.

Overall, DRYAD_{dec} allows seven categories of recursive functions or predicates with various types, constraints on their definitions and forms of occurrence in a formula. Figure 3 gives several common examples of recursive definitions expressible in DRYAD_{dec}. We explain the intuition behind each category below:

Increasing or decreasing Int function¹ defines the maximum or minimum value of $it[x]$, where x is the location being unfolded in the tree. The local term $it[x]$ is an integer term defined only based on the local data fields of x . The most common example is $it[x] = x.\text{key}$; then the function gives the maximum or minimum **key** stored in a tree. These increasing/decreasing functions can be combined using standard arithmetic connectives to form atomic formulae.

Increasing IntSet function defines the union of all set terms $ST[x]$ for any location x under the tree, where $ST[x]$ is a set of local integer terms defined only based on the local data fields of x . The most typical example is the function representing the set of all keys w.r.t. the data field **key**, where $ST[x] = \{x.\text{key}\}$. These *IntSet* functions can be combined with regular *Int* terms arbitrarily to form *IntSet* terms in DRYAD_{dec}, which can be further used to construct atomic

¹ Intuitively, a DRYAD_{dec} function is increasing/decreasing if its value monotonically increases/decreases when the input tree expands. The monotonicity will be formally defined in Section 3.1

formulae for set-inclusion and subset relationship. The only restriction is that the subset checking $S_1 \subseteq S_2$ can occur *negatively only*.

There are two types of **measure functions**. Intuitively, they recursively define Max- and Sum-based measurements of a tree or tree segment, respectively. For each node x under the tree, it counts towards the measurement, i.e., the height/size being increased by 1, if and only if a local formula $v[x]$ is satisfied. In Figure 1, this conditional value is written as $\text{ite}^r(v[x], 1, 0)$, where r is an integer constant called *intermittence*. For example, the black height for red black trees can be defined with intermittence 2: $\text{ite}^2(x.\text{color} = \text{black}, 1, 0)$. The intermittence’s semantics will be explained in Section 2.3. Specifically, when $v[x] \stackrel{\text{def}}{=} \text{true}$ and $r = 1$, the corresponding Max- and Sum-based functions define the regular tree height and size, respectively. In this paper, we denote them as height^* and size^* .

A measure-related *Int* term can be a measure function $f^*(x)$ only, or a difference of form $f^*(x_1) - f^*(x_2)$. A measure-related *Int* term can be compared with a constant K . For example, one can specify two trees with the same height using $\text{height}^*(x_1) - \text{height}^*(x_2) = 0$.

General predicate is satisfied by trees (x) if and only if a local constraint φ is satisfied between any location in x . Notice that φ may involve other non-measure-related functions or predicates (with some restrictions as shown in Figure 1). For example, the sorted^* property can be defined based on max^* and min^* (see the definition of sorted^* in Figure 3).

Measure-related predicate is similar to general predicates. In addition to everything allowed in the definition of general predicates, a measure-related predicate is allowed to involve a single measure-related function in the difference form. For example, an avl^* -tree requires the height^* -difference between two subtrees is at most one (see the definition of avl^* in Figure 3).

2.2 Syntactic Restrictions for Decidability

As we have mentioned before, the syntax of $\text{DRYAD}_{\text{dec}}$ is carefully crafted for decidability. Besides the specific syntactical restrictions delineated above for the definitions in each category of recursive functions or predicates, $\text{DRYAD}_{\text{dec}}$ also restricts how variables, functions and predicates can be related to each other. As shown in Figure 2, a variable x is considered related to a measure function if x occurs in a measure-related predicate or in the difference form $f^*(x) - f^*(y)$. One important restriction of $\text{DRYAD}_{\text{dec}}$ is that a location variable can be related to only one measure function. For example, $\text{DRYAD}_{\text{dec}}$ cannot express a single-path tree: $\text{height}^*(x) = \text{size}^*(x)$.

Insight Behind the Syntax. Intuitively, the $\text{DRYAD}_{\text{dec}}$ syntax characterizes the class of formulae *independent* to the height/size of the tree. Hence non-measure functions such as min^* or max^* can occur unrestrictedly in the logic, as their values are only determined by the “witness nodes”. For measure functions such as height or size, obviously they are determined by the height/size of the tree; that’s why we allow only differences between measure functions such as

$height^*(x_1) - height^*(x_2)$, as the difference is *unchanged* if we tailor both the two trees rooted by x_1 and x_2 at the same time. Likewise for subset relation, the negation of subset relation $S_1 \not\subseteq S_2$ can also be captured by a “witness node” which is in the set of S_1 but not in the set of S_2 whereas the subset relation $S_1 \subseteq S_2$ is determined by all elements in two sets. Therefore, $S_1 \not\subseteq S_2$ is allowed whereas $S_1 \subseteq S_2$ is not as $S_1 \subseteq S_2$ is not ensured to be unchanged through tailoring. To conclude, we try to maximize the logic without losing decidability.

Capabilities And Limitations. DRYAD_{dec} can express all standard tree-based data structures such as lists, trees, lists of trees, etc., and some limited non-tree data structures such as doubly linked lists or cyclic lists. However, DRYAD (and inherently DRYAD_{dec}) is unable or not natural to express non-tree data structures, e.g., DAGs or overlaid data structures. The main restrictions from DRYAD to DRYAD_{dec} are twofold. First, only Max- and Sum-based measure functions are allowed. For example, DRYAD_{dec} cannot define the length of the leftmost path of a tree. Second, properties involving multiple measure functions are not allowed. For example, as red-black trees are defined using black-height, DRYAD_{dec} cannot describe the real height of a red-black tree.

Category	Name	Definition
Measure Function (Max-based)	$height^*$	$\text{ite}(\text{isNil}(x), 0, \max(height^*(x.left), height^*(x.right)) + 1)$
	bh^*	$\text{ite}(\text{isNil}(x), 0, \max(bh^*(x.left), bh^*(x.right)) + \text{ite}^2(x.isBlack, 1, 0))$
Measure Function (Sum-based)	$size^*$	$\text{ite}(\text{isNil}(x), 0, size^*(x.left) + size^*(x.right) + 1)$
Non-Measure Function	max^*	$\text{ite}(\text{isNil}(x), -\infty, \max(max^*(x.left), max^*(x.right), x.key))$
	min^*	$\text{ite}(\text{isNil}(x), \infty, \min(max^*(x.left), min^*(x.right), x.key))$
	$keys^*$	$\text{ite}(\text{isNil}(x), \emptyset, keys^*(x.left) \cup keys^*(x.right) \cup \{x.key\})$
Measure-related Predicate	avl^*	$\text{ite}(\text{isNil}(x), \text{true}, avl^*(x.left) \wedge avl^*(x.right) \wedge 1 \geq height^*(x.left) - height^*(x.right) \geq -1)$
	rbt^*	$\text{ite}(\text{isNil}(x), \text{true}, rbt^*(x.left) \wedge rbt^*(x.right) \wedge bh^*(x.left) = bh^*(x.right))$
General Predicate	$sorted^*$	$\text{ite}(\text{isNil}(x), \text{true}, sorted^*(x.left) \wedge sorted^*(x.right) \wedge max^*(x.left) < x.key < min^*(x.right))$
	$treap^*$	$\text{ite}(\text{isNil}(x), \text{true}, treap^*(x.left) \wedge treap^*(x.right) \wedge max_key^*(x.left) < x.key < min_key^*(x.right) \wedge max_prt^*(x.left) < x.prt \wedge max_prt^*(x.right) < x.prt)$

Fig. 3: List of recursive definitions

2.3 Semantics

The semantics of DRYAD_{dec} is consistent with the semantics of DRYAD defined in [26], which is interpreted on program heaps. A heap consists of a finite set of locations with the same layout. Each location contains a set of pointer fields Dir and a set of data fields DF . In addition, there is a set of location variables LV , a set of integer variables IV , and a special location nil where the pointer fields can point to. We call $\Sigma = (Dir, DF, LV, IV)$ a signature for the DRYAD_{dec} logic, and call the heap w.r.t. Σ a Σ -heap. The formal definition is as below:

Definition 1. Let $\Sigma = (Dir, DF)$. A Σ -heap is a tuple (N, pf, df) where:

- N is a finite set of locations; $nil \in N$ is a special location;
- $pf: (N \setminus \{nil\}) \times Dir \rightarrow N$ is a function defining the pointer fields;
- $df: (N \setminus \{nil\}) \times DF \rightarrow \mathbb{Z}$ is a function defining the data fields. □

A recursive definition $f^*(x)$ can be interpreted on a Σ -heap (N, pf, df) by mapping x to a location n_x in the heap. As f^* is a recursive definition, $f^*(x)$ is *undefined* if n_x is not the root of a tree; otherwise it is evaluated inductively using the recursive definition of f^* . Notice that the evaluation is only determined by a subset of N that is reachable from n_x . If a heap T 's locations form a tree, we use $f^*(T)$ to represent the interpretation of $f^*(x)$ with x mapped to the root of T . We simply call T a Σ -tree. We denote n as $root(T)$, and the subtree rooted by $n.dir$ as $T.dir$.

A DRYAD_{dec} formula $\varphi(\bar{x}, \bar{j}, \bar{r})$ can be interpreted on a Σ -heap by mapping every *Loc* variable in \bar{x} to a location in the heap and mapping every *Int* variable in \bar{j} and *IntSet* variable in \bar{r} to the corresponding sort. The mapping is valid only if every *Loc* variable maps to the root of a tree in the heap; otherwise the interpretation is undefined.

Most logical connectives and recursive functions/predicates are interpreted as one can expect. In addition, measure functions have a special intermittence constraint. Recall that any measure function f^* 's definition comes with an intermittence r occurred in form of $\mathbf{ite}^r(v[x], 1, 0)$. The intermittence is a positive integer indicating *how often* the local formula $v[x]$ should be satisfied in the trees. Formally, f^* is defined on a tree T only if the following intermittence constraint is satisfied: for *any* node x in T and its $(r - 1)$ immediate ancestors, there is a node w within these r nodes such that $v[w]$ is true.

Notice that a satisfiable φ with m *Loc* variables x_1, \dots, x_m can always be satisfied by a heap consisting of m disjoint trees T_1, \dots, T_m by mapping every x_i to the root of T_i . In the rest of the paper, we focus on checking satisfiability and consider only these disjoint-tree models.

3 Proof of Decidability

In this section, we prove that the satisfiability problem of DRYAD_{dec} is decidable. The crux of the proof is the *small model property*: Given a DRYAD_{dec} formula φ , it is satisfiable only if it is satisfied by a model of bounded size. The main idea is to show that if φ is satisfied by a model larger than the bound, one can tailor the model to obtain a smaller model which preserves the satisfiability (Theorem 1).

Intuitively, the value of an increasing/decreasing *Int* function or increasing *IntSet* function always relies on a *witness node*. For example, if an increasing *Int* function mif^* is defined w.r.t. a local term it within any tree T , there is a witness node w s.t. $mif^*(T) = it[w]$ and $it[w] \geq it[u]$ for any other node u . Then these function values can be preserved as long as these witness nodes are retained in the tailored model (Lemma 6).

$d : Dir $	$n : \# \text{ Int Variables}$	$m : \# \text{ Loc Variables}$
$P : \# \text{ General Predicates}$	$M : \# \text{ } lif^* \text{-related Predicates}$	$C : \text{ Balance Bound}$
$D_{ht} : \text{ Height Bound}$	$D_{sz} : \text{ Size Bound}$	$D_{sub} : \text{ Subtractive Bound}$
$F : \# \text{ Increasing/Decreasing Int Fuctions}$	$E : \# \text{ Increasing IntSet Fuctions}$	

Fig. 4: Denotations for metrics

The most challenging part is that the value of a measure-related function will become smaller. Nonetheless, we prove that one can tailor the tree appropriately such that the height/size is reduced by exactly 1 while all relevant recursive predicates are still preserved. Then as these measure functions only occur in the form $f^*(x_1) - f^*(x_2)$, both $f^*(x_1)$ and $f^*(x_2)$ will be reduced by 1 simultaneously and the difference will remain unchanged. Moreover, we prove the tailoring guarantees that the evaluation of other functions and predicates are not affected (Lemmas 7 and 8).

3.1 Preliminaries

We start with some formal definitions and lemmas. The proofs for these lemmas can be found at the project website [1].

Normalization. We normalize a $DRYAD_{dec}$ formula φ through repeatedly applying the following steps until no rule can be applied:

1. For every **ite**-expression $E_{ite} = \mathbf{ite}(l, t_1, t_2)$ in φ , rewrite φ to $(l \wedge \varphi[t_1/E_{ite}] \vee (\neg l \wedge \varphi[t_2/E_{ite}]))$;
2. For every literal $S_1 \not\subseteq S_2$, introduce a fresh integer variable w as a witness, and replace the literal with $w \in S_1 \wedge w \notin S_2$;
3. For every atomic formula of the form $t \in A \cap B$ or $t \in A \cup B$, replace it with $t \in A \wedge t \in B$ or $t \in A \vee t \in B$, respectively;
4. For every atomic formula of the form $t_1 \in \{t_2\}$, replace it with $t_1 = t_2$;
5. For every atomic formula $t \in S$ where t is a non-variable expression, introduce a fresh integer variable j and replace $t \in S$ with $j \in S \wedge j = t$;
6. For every literal $lif^*(x) - lif^*(y) \not\geq K$ or $eif^*(x) - eif^*(y) \not\geq K$, replace it with $lif^*(y) - lif^*(x) \geq 1 - K$ or $eif^*(y) - eif^*(x) \geq 1 - K$.

We denote the normalized formula constructed from φ as $Norm(\varphi)$. The first two steps remove the **ite**-expressions and the $\not\subseteq$ relations from the formula. Steps 3–5 make sure that set terms occur in the form of $j \in sf^*(x)$ only. Step 6 makes sure differences between measure functions occur positively only. To check the satisfiability of φ , one can always normalize the formula first, as the normalization process preserves satisfiability, which can be trivially proved:

Lemma 1. *For any $DRYAD_{dec}$ formula φ , φ and $Norm(\varphi)$ are equisatisfiable.*

Formula Metrics. The size bound for the small model property will be determined by a set of metrics regarding the signature Σ , the formula φ and the set of recursive definitions it relies on. For the rest of the paper, we fix the denotation for these metrics, as shown in Figure 4. Besides simple counting of functions or predicates, these metrics also include the bounds on various kinds of constants involved in the formula. Specifically, we define the following four bounds:

Definition 2 (Balance Bound). For any Max-based measure function lif^* , the balance bound C is the maximal constant in the set: $\{\mathbf{ite}(K > 0, K, 1 - K) \mid lif^*(t) - lif^*(t') \geq K \text{ occurred in the definition of a } lif^* \text{-related predicate}\}$.

Definition 3 (Subtractive Bound). The subtractive bound D_{sub} of a formula φ is the maximal constant in the set: $\{\mathbf{max}(K, 0) \mid lif^*(x) - lif^*(y) \geq K \text{ or } eif^*(x) - eif^*(y) \geq K \text{ occurred positively in } \varphi\}$.

Definition 4 (Height Bound). The height bound D_{ht} of a formula φ is the maximal constant in the set: $\{rK \mid lif^*(y) \geq K \text{ occurred positively in } \varphi \text{ and } r \text{ is the intermittence of } lif^*\}$.

Definition 5 (Size Bound). The size bound D_{sz} of a formula φ is the maximal constant in the set: $\{(\frac{d^r-1}{d-1}) \cdot K + 1 \mid eif^*(y) \geq K \text{ occurred positively in } \varphi \text{ and } r \text{ is the intermittence of } eif^*\}$.

Remark: For all of the above bounds, if the corresponding set is empty, we define the bound to be 0.

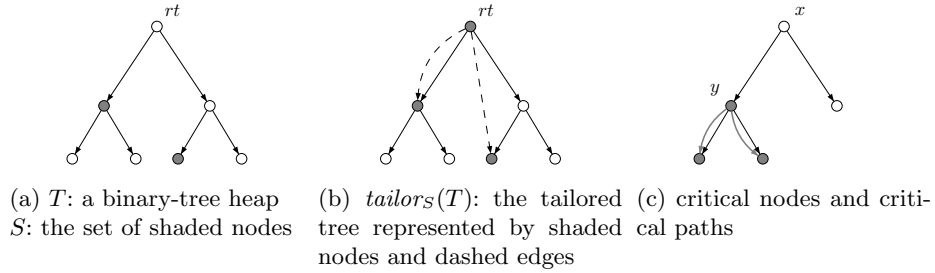


Fig. 5: A binary tree example of tailored trees and critical nodes and paths

Tailored Tree and Monotonicity. As a key concept in the decidability proof, the small model is formalized via *tree tailoring*: a tree model can be tailored to obtain a smaller model.

Definition 6 (Tailored tree). Let $T = (N, pf, df)$ be tree, and let $S \subset N$ be a subset, then the tailored tree $tailors_S(T)$ can be defined as (N', pf', df') , where (i) $N' = S \cup \{lca(S') \mid S' \subseteq S\}$ where $lca(S')$ is the lowest common ancestor of S' ; (ii) $pf'(x, dir) = lca(N' \cap T_x.dir)$ for any $x \in N'$ and $dir \in Dir$, where $T_x.dir$ is the subtree of T rooted by $x.dir$; and (iii) $df' = df|_{N' \times DF}$.

Note that N' is LCA-closed, the lowest common ancestor $\text{lca}(N' \cap T.\text{dir})$ defined by $pf(x, \text{dir})$ always belongs to N' . As an example, Figure 5a shows a tree-shaped heap T and a subset S of nodes (the shaded ones); Figure 5b shows the tailored tree $\text{tailor}_S(T)$ constructed from S . The edges of the tailored tree are represented using dashed edges.

Now with tailored tree formally defined, we can prove the *monotonicity* of non-measure functions/predicates, a very important property for our decidability proof. We prove the following three lemmas.

Lemma 2 (Monotonicity for increasing/decreasing function). *Let mif^* (or mdf^*) be an increasing (or decreasing) function w.r.t. Σ . Let T be a Σ -tree and let $\text{tailor}_S(T)$ be the tailored tree w.r.t. a subset of nodes S . Then $\text{mif}^*(T) \geq \text{mif}^*(\text{tailor}_S(T))$ (or $\text{mdf}^*(T) \leq \text{mdf}^*(\text{tailor}_S(T))$).*

Lemma 3 (Monotonicity for increasing *IntSet* function). *Let T be a Σ -tree and let $\text{tailor}_S(T)$ be the tailored tree w.r.t. a subset of nodes S . Then for any increasing set function sf^* , $\text{sf}^*(\text{tailor}_S(T)) \subseteq \text{sf}^*(T)$.*

Lemma 4 (Monotonicity for general predicate). *Let T be a Σ -tree and let $\text{tailor}_S(T)$ be the tailored tree w.r.t. a subset of nodes S . Then for any general predicate gp^* , $\text{gp}^*(T)$ implies $\text{gp}^*(\text{tailor}_S(T))$.*

Critical Path. While measure-related functions/predicates do not have witness nodes, their evaluation can be determined by a set of paths, which we call *critical paths*.

Definition 7 (Critical Node and Critical Path). *Let $T = (N, pf, df)$ be a nonempty Σ -tree and $y \in N$ be a node. Let lif^* be a Max-based measure function. Then y is a critical node of T w.r.t. lif^* if one of the following conditions holds:*

1. $\text{lif}^*(y) \geq \text{lif}^*(z)$ for any other sibling node z ;
2. there is a measure-related predicate mp^* whose recursive definition involves a subformula of the form $\text{lif}^*(x.\text{dir}_1) - \text{lif}^*(x.\text{dir}_2) \geq K$, and there is a node $x \in N$ such that:
 - either $K \geq 1$, $\text{lif}^*(x.\text{dir}_1) - \text{lif}^*(x.\text{dir}_2) = K$, $y = x.\text{dir}_2$ and $x.\text{dir}_1$ is a critical node;
 - or $K \leq 0$, $\text{lif}^*(x.\text{dir}_1) - \text{lif}^*(x.\text{dir}_2) = K - 1$, $y = x.\text{dir}_1$ and $x.\text{dir}_2$ is a critical node.

For the second case, we also call y a critical child of x . Moreover, a critical path w.r.t. lif^* is a path from a child of T to a leaf consisting of critical nodes only.

As an example, Figure 5c shows a binary tree rooted by x . The shaded nodes are critical nodes and curved edges are two critical paths w.r.t. height^* . (See definition of height^* in Figure 3.)

Lemma 5 (Length bound for critical paths). *Let lif^* be a Max-based function with intermittence r and with a local constraint v , let T be a d -ary tree. Then for any critical path of T w.r.t. lif^* , the number of nodes satisfying v on the path is at least $\lfloor \frac{\text{lif}^*(T) - 1}{(d-1)Cr + 1} \rfloor$, where C is the balance bound of lif^* .*

3.2 Tailorability

The tailorability of various functions/predicates is the crux of guaranteeing the small model property, which in turn guarantees the decidability. As mentioned before, non-measure functions/predicates can be easily preserved as long as the tailoring does not affect witness nodes.

Lemma 6 (Tailorability for non-measure functions and general predicates). *Let $T = (N, pf, df)$ be a tree, $S \subset N$ be a subset of nodes.*

Then if the height of T is greater than $P + F + |S|$, there is a tailored tree T' of T such that

- (i) T' contains all nodes of S ;
- (ii) $f^*(T') = f^*(T)$ for any increasing/decreasing Int function f^* ;
- (iii) $gp^*(T') \leftrightarrow gp^*(T)$ for any general predicate gp^* .

Proof. See [1]. □

For a Max-based function, a large tree can be tailored by removing exactly one node from every critical path; hence the function value is reduced by 1. Similarly, Sum-based functions can also be reduced by 1 through tailoring.

Lemma 7 (Tailorability for Max-based function). *Let $T = (N, pf, df)$ be a d -ary tree, $S \subset N$ be a subset of nodes. Let lif^* be a Max-based measure function with intermittence r and balance bound C . Then if $lif^*(T) > (P + M + F + |S| + 1) \cdot ((d - 1)Cr + 1)$, there is a tailored tree T' of T such that*

- (i) T' contains all nodes of S ;
- (ii) $f^*(T') = f^*(T)$ for any increasing/decreasing Int function f^* ;
- (iii) $gp^*(T') \leftrightarrow gp^*(T)$ for any general predicate gp^* ;
- (iv) $lif^*(T') = lif^*(T) - 1$;
- (v) $mp^*(T') \leftrightarrow mp^*(T)$ for any lif^* -related predicate mp^* .

Proof. Let the definition of lif^* be $\text{ite}(\text{isNil}(x), 0, \dots + \text{ite}^r(v[x], 1, 0))$. Consider an arbitrary critical path w.r.t. lif^* in T . By Lemma 5, the number of nodes in the path satisfying the local constraint v from the definition of lif^* is at least

$$\lfloor \frac{lif^*(T) - 1}{(d - 1)Cr + 1} \rfloor \geq \lfloor \frac{(P + M + F + |S| + 1) \cdot ((d - 1)Cr + 1)}{(d - 1)Cr + 1} \rfloor = P + M + F + |S| + 1$$

Let \mathcal{N} be the set including all these nodes. We denote a node in \mathcal{N} as n_j if it is the j -th highest one in the set. For each j , consider the set of nodes $\mathcal{N}_j \stackrel{\text{def}}{=} \{n \mid n \prec n_j \wedge n \not\prec n_{j+1}\}$, where $n \prec n_j$ denotes that n is a descendant of n_j . Intuitively, \mathcal{N}_j is the root or a descendant of a sibling of n_{j+1} . Notice that there are at least $P + M + F + |S| + 1$ such sets and they are all disjoint, i.e., there is a set of at least $P + M + F + 1$ nodes such that for every node j in the

set, $\mathcal{N}_j \cap S = \emptyset$. Furthermore, consider the witness node for every $f^*(T)$, where f^* is an increasing or decreasing *Int* function, among the remaining at least $P+M+F+1$ nodes, at least $P+M+1$ ones are nodes for which corresponding set \mathcal{N}_j does not contain any witness nodes. Moreover, as the number of all predicates is $P+M$, there is at least one node l such that n_l and n_{l+1} ² agree on the evaluation of all general predicates and lif^* -related predicates.

Now we can replace the subtree rooted by n_l with the subtree rooted by n_{l+1} to form a tailored tree T_l . Notice that T_l holds the first three properties for the desired tailored tree:

1. T_l retains all nodes of S , as $\mathcal{N}_j \cap S = \emptyset$.
2. $f^*(T_l) = f^*(T)$ for any increasing or decreasing f^* .
3. $gp^*(T_l)$ if and only if $gp^*(T)$ for any general predicate gp^* .

The reason for properties (i) and (ii) to hold is straightforward. For property (iii), consider three situations:

1. if $gp^*(T)$ is true, so is $gp^*(T_l)$ by Lemma 4.
2. if $gp^*(T)$ is false and $gp^*(n_l)$ is true, then T does not satisfy gp due to a path not affected by the tailoring. Hence $gp^*(T_l)$ remains false.
3. if $gp^*(T)$ is false and $gp^*(n_l)$ is false, by our assumption about l , n_l and n_{l+1} agree on the evaluation of all general predicates. Hence $gp^*(n_{l+1})$ is also false. Then by Lemma 4, $gp^*(T_l)$ is also false.

Moreover, as n_l and n_{l+1} agree on all predicates, the tailoring also preserves any lif^* -related predicate mp .

This tailoring also removes exactly one node from \mathcal{N} for the critical path we are considering. One can continue this tailoring for other critical paths until all critical paths have been shortened and the value of lif^* is reduced by 1. We claim that the resulting tree is just the desired tailored tree T' . As each tailoring guarantees the first three properties, we only need to show the last two properties. Property (iv) is obvious: all critical paths of z are shortened and $lif^*(z)$ is reduced by 1. For Property (v), we prove it by a bottom-up induction for any node z under which a tailoring took place. The evaluation of any lif^* -related predicate $mp^*(z)$ is not affected: if the subtree under z replaced another subtree rooted by z' , $mp^*(z)$ if and only if $mp^*(z')$ is true; otherwise, there was a separate tailoring for each critical child of z . Therefore

- by induction hypothesis, $mp^*(z.dir)$ is preserved for any mp^* and any dir ;
- local *Int* terms are not affected, as z is unchanged during the tailoring;
- for any increasing or decreasing function f^* and any child $T.dir$, the value of $f^*(T.dir)$ is preserved during every tailoring and still unchanged;
- similarly, $gp^*(T.dir)$ for any general predicate gp^* is unchanged;
- for any critical child $T.dir$, $lif^*(T.dir)$ only occurs in subtractive formulae in the recursive definition for $lif^*(x)$. Notice that $lif^*(T.dir)$ is decreased by 1 and so is any other critical $lif^*(T.dir')$, the evaluation of these subtractive formulae will be unaffected.

² Let n_{l+1} be *nil* if $|\mathcal{N}| \leq l$.

□

Lemma 8 (Tailorability for Sum-based function). *Let $T = (N, pf, df)$ be a d -ary tree, $S \subset N$ be a subset of nodes. Let EIF^* be a Sum-based measure function with intermittence r . Then if $EIF^*(T) > 2 \cdot (|S| + F + 2^P) - 1$, there is a tailored tree T' of T such that*

- (i) T' retains all nodes of S ;
- (ii) $f^*(T') = f^*(T)$ for any increasing/decreasing *Int* function f^* ;
- (iii) $gp^*(T') \leftrightarrow gp^*(T)$ for any general predicate gp^* ;
- (iv) $EIF^*(T') = EIF^*(T) - 1$;

Proof. Let the definition of EIF^* be $\text{ite}(\text{isNil}(x), 0, \dots + \text{ite}^r(v[x], 1, 0))$. Let \mathcal{N} be the set including all nodes satisfying v . Note that $|\mathcal{N}| = EIF^*(T) \geq 2 \cdot (|S| + F + 2^P)$. Consider those nodes in \mathcal{N} but not above two other nodes in \mathcal{N} from two different branches: $\mathcal{N}' \stackrel{\text{def}}{=} \{n \in \mathcal{N}, \exists n_1, n_2, dir_1, dir_2 : dir_1 \neq dir_2 \wedge n_1 \prec n.dir_1 \wedge n_2 \prec n.dir_2\}$. Similar to the proof of Lemma 7, for each node $n \in \mathcal{N}'$, T can be tailored by removing the subtree rooted by n or replaced with its subtree preserving all nodes from \mathcal{N}' . We denote the set of removed nodes \mathcal{N}_n . Moreover, it is not hard to see that $|\mathcal{N}'| \geq \lceil \frac{|\mathcal{N}|+1}{2} \rceil \geq |S| + F + 2^P + 1$.

Now we remove from \mathcal{N}' every node n such that $\mathcal{N}_n \cap S \neq \emptyset$ or \mathcal{N}_n contains the witness node for $f^*(T)$ for a increasing or decreasing *Int* function f^* . Let the set of the remaining nodes in \mathcal{N}' be \mathcal{N}'' . As the number of removed nodes from \mathcal{N}' is at most $|S| + F$, $|\mathcal{N}''| \geq 2^P + 1$. Therefore there are at least two nodes $n_1, n_2 \in \mathcal{N}''$ such that n_1 and n_2 agree on all general predicates. If n_1 and n_2 are on the same path and n_1 is above n_2 , then we tailor \mathcal{N}_{n_1} ; otherwise we tailor \mathcal{N}_{n_2} . WLOG, assume the tailoring replaces n_2 with n'_2 and forms T' . The tailoring satisfies all desired properties:

1. T' retains all nodes of S as \mathcal{N}_{n_2} does not contain any node of S .
2. T and T' agree on all increasing/decreasing functions as all witness nodes are retained.
3. T and T' also agree on all general predicates: for any gp^* , if n_2 and n'_2 agree on gp^* , the preservation can be propagated up to the root of T . Otherwise, $gp^*(n_2)$ is false and $gp^*(n'_2)$ is true. Notice that n_1 and n_2 are not on the same path in this situation – otherwise n'_2 is between n_2 and n_1 and does not satisfy gp^* . Then n_1 is not affected by the tailoring and $gp^*(n_1)$ remains false and propagates up to the root: $gp^*(T)$ remains false.
4. By the definition of \mathcal{N} , n_2 is the only node in \mathcal{N}_{n_2} that satisfies the local constraint ϵ ; hence $EIF^*(T') = EIF^*(T) - 1$.

□

3.3 Decidability

Now we are ready to show the small model property for DRYAD_{dec} .

Theorem 1. *Let φ be a Σ -formula in DRYAD_{dec} . Then there is a height bound h_φ such that φ is satisfiable if and only if it can be satisfied by trees with height at most h_φ .*

Proof. According to Lemma 1, we assume φ is normalized and satisfiable. Consider any m disjoint trees T_1 through T_m satisfying φ . For any T_i , we construct a subset of nodes S_i as follows: for every literal $j \in sf^*(x_i)$ where $sf^*(x)$ is an *IntSet* function recursively defined as $\text{ite}(\text{isNil}(x), \emptyset, (\bigcup_{dir} sf^*(x.dir)) \cup ST[x])$, there must be a witness node y such that $j \in T_i[y]$. We add y to S_i . For a fixed location variable x_i , there are up to En atomic formulae of the form $j \in sf^*(x_i)$. Hence there are up to En nodes in the subset S_i constructed for T_i .

Now if x_i is related to a Max-based measure function, we claim the following height bound: $h_\varphi = (En + P + M + F + 1) \cdot ((d-1)Cr + 2) + D_{ht} + (m-1)D_{sub} - 1$. for a set of variables J including x_i . We define J recursively as the smallest set satisfying the following properties:

- x_i belongs to J ;
- if $lif^*(x_1) - lif^*(x_2) \geq K$ occurs in φ and the inequation is tight, i.e., the model we are considering satisfies $lif^*(x_1) - lif^*(x_2) = K$, then x_2 belongs to J if x_1 does.

Similarly, if x_i is related to a Sum-based measure function ef^* , we claim the following size bound: $U_\varphi = 6(En + F + 2^P) - 3 + 2D_{sz} + 2(m-1)D_{sub}$. Note that the size bound is trivially a height bound as well. The proofs for the two bounds h_φ and U_φ can be found at [1].

If x_i is not related to any measure function, we claim a height bound $En + P + F$. When T_i 's height is greater than the bound, by Lemma 6, it can be tailored to T'_i and have all set-inclusions, non-measure *Int* functions and general predicates preserved.

Now we obtain a tree T'_i with strictly fewer nodes. By assumption, T_i is the smallest model and T'_i should not satisfy φ . In the rest of the proof, we will show they do satisfy φ ; and the contradiction concludes the proof.

As φ is quantifier-free, we only need to show that for any literal in φ , if T_i satisfies it, so does T'_i . We prove this for each type of literals:

Measure-related Predicate. For any measure-related predicate $mp^*(x_j)$ in φ , x_j must be involved in a Max-based measure function lif^* or not involved in any measure function. Replacing T_j with T'_j guarantees that $lif^*(T'_j) = lif^*(T_j) - 1$, and according to Lemma 7, $mp^*(T_j) = mp^*(T'_j)$.

Measure-related Inequation. For any atomic formula $f^*(x_i) - f^*(y) \geq K$ affected by the tailoring, the second rule for the construction of J guarantees that x_i is in J . If $f^*(x_i) - f^*(y)$ is strictly greater than K or less than K , as the value of $f^*(x_i)$ is reduced by only 1 in the course of shrinking, the inequation is still satisfied or unsatisfied. Otherwise, $f^*(x_i) - f^*(y) = K$, then y is also contained

in J . In that case, $f^*(x_i)$ is also reduced by 1. Hence $f^*(x_i) - f^*(y) \geq K$ will remain satisfied or unsatisfied in T'_i .

For any atomic formula $f^*(x) \geq K$ affected by the tailoring, the tailoring only happens when $f^*(x)$ is greater than K before the tailoring. We have shown above that $f^*(x) \geq K$ is still satisfied after each tailoring. Hence the satisfiability is preserved.

For any atomic formula $f^*(y) \leq K$, the tailorings will make it easier to be satisfied.

Non-measure predicate or function. By Lemma 7 and Lemma 8, any tailoring described above does not affect the evaluation of any non-measure predicate or function, including any general predicate and increasing/decreasing function.

Set Inclusion. For any $j \in sf^*(x_j)$ in φ satisfied by T_j , if it occurs positively, the witness node is in S and will be preserved during the tailoring from T_j to T'_j ; hence it is satisfied by T'_j as well. If T_j does not satisfy $j \in sf^*(x_j)$, as the set $sf^*(x_j)$ becomes smaller during the tailoring (by Lemma 3), T'_j does not satisfy $j \in sf^*(x_j)$.

isNil predicate and other boolean variables. These are not affected by tree tailoring and obviously unchanged. □

Corollary 1. *The satisfiability problem of DRYAD_{dec} is decidable. For a fixed signature Σ and a fixed set of recursive functions, the problem is in NEXPTIME .*

Proof. Given a DRYAD_{dec} formula φ with maximum constant bound D (including subtractive, size and height bounds), by Theorem 1, a minimal satisfying model of the normalized formula consists of m disjoint trees, each of which has a bounded height $\mathcal{O}(n + mD)$, i.e., there are up to $2^{\mathcal{O}(n+mD)}$ nodes in the smallest model. Hence one can unfold every recursive function/predicate in the formula for $2^{\mathcal{O}(n+mD)}$ times and leave them uninterpreted. The resulting formula is equisatisfiable with φ and obviously decidable as it is in the theory of quantifier-free uninterpreted functions and linear integer arithmetic (QF_UFLIA), which is NP-complete. As the size of the QF_UFLIA formula is $2^{\mathcal{O}(n+mD)}$, the satisfiability of DRYAD_{dec} is decidable and is in NEXPTIME .

If Σ does not involve any Max-based measure function, then the size of the tree and the QF_UFLIA formula is bounded by $\mathcal{O}(n + mD)$, and the time complexity becomes NP-complete. □

4 Experiments

To demonstrate the expressivity of DRYAD_{dec} and the efficiency of the decision procedure, we implemented the decision procedure and solved 220+ DRYAD_{dec} formulae. These formulae encode various problems from three verification/synthesis scenarios: natural proof verification, fusion of recursive tree traversals, and synthesis of CLIA functions. The implementation is SMT-based: for each

formula, we first analytically computed the height bound; then the decision procedure encoded the DRYAD_{dec} formula to a QF_UFLIA formula with the computed bound, and invoked an SMT solver to solve the formula.

Applications. The first set of 61 DRYAD_{dec} formulae is for program verification. We aim to verify the functional correctness of five tree-manipulating programs, i.e., every routine should ensure that the returned tree after insertion remains a corresponding data-structure. We have described `insertToLeft` in Section 1; `BST-insert`, `Treap-insert`, `AVL-insert` and `RBT-insert` are self-explanatory. We manually broke down each program into basic blocks and wrote all of the Natural Proof Verification Conditions (NPVC) following the NPVC-generation algorithm adapted from [26]. For sanity checking, we also manually implanted some artificial bugs to the programs and created the corresponding NPVCs.

The second set of 48 formulae is for checking the fusibility of recursive tree traversals. Fusion of tree traversals arises in numerous settings [37,31,44,43,17,18,47,27,8] for performance concern. One of the crucial parts for this fusion process is to check the *fusibility* of two traversals, i.e., if there exists a fused traversal that has identical behavior with the original two traversals. We used DRYAD_{dec} to check all possible fusions of two pairs of traversals: a pair of height-based, mutually recursive traversals and another pair of a post-order traversal execute before a pre-order traversal. Neither can be handled by state-of-the-art checkers [48]. Please find more details of encoding fusibility to DRYAD_{dec} at [1].

The last set of 112 formulae is for synthesizing Conditional Linear Integer Arithmetic (CLIA) functions. The goal is to synthesize a sequence of arithmetic operations that implements an unknown function described by a formula. DRYAD_{dec} formulae are created by our in-house Syntax Guided Synthesis SyGuS synthesizer [13] as queries raised from the Counter-Example Guided Inductive Synthesis (CEGIS) algorithm. We adopted 23 benchmarks from the 2017 SyGuS [2] competition, for which the queries fall into DRYAD_{dec} . The detail of the CEGIS algorithm and the DRYAD_{dec} encoding can be found at [1].

Scenario	Signature	E	P	M	F	r	C	D_{sub}	Bound
BST mutation	$bst^*, keys^*, max^*, min^*$	1	1	0	2	0	0	0	$n + 3$
Treap mutation	$treap^*, prts^*, max_prt^*$ $keys^*, max_key^*, min_key^*$	2	1	0	3	0	0	0	$n + 4$
AVL mutation	$height^*, avl^*$	0	0	1	0	1	2	3	$3m - 2$
RBT mutation	bh^*, rbt^*	0	0	1	0	2	1	3	$3m - 2$
CLIA	$\{exp_{spec_f, F}^* \mid \emptyset \subset F \subseteq G\}$	0	$2^{ G } - 1$	0	0	0	0	0	$ G \cdot spec_f $
Fusion	$dp^*, schd^*$	0	2	0	F	0	0	0	$F + 2$

Table 1: Height/Size bounds for different scenarios (Metrics defined in Figure 4)

Bound Optimization. We implemented the decision procedure with a set of optimizations. The height/size bound derived in Theorem 1 is general and loose, affecting the decision procedure’s scalability. We developed many optimization strategies for different situations. Every strategy is automatically applied when the corresponding condition is satisfied. Table 1 shows the best bounds we obtain

for each scenario after all applicable optimizations. Below we explain the main optimization strategies we developed.

To check the satisfiability of a formula φ , we first converted φ to the Disjunctive Normal Form (DNF) and computed the height/size bound for each disjunct separately, as φ is satisfiable if and only if one of the disjuncts is satisfiable. This helps us compute a better bound in many situations, as for each disjunct, at least one or more factors used in the bound computation, e.g., n , m , D_{ht} , D_{sz} and D_{sub} , can be reduced.

Analyzing how variables occur in φ can also be helpful. For example, the number of location variables m only contribute to the bound with the term $(m-1)D_{sub}$. This term is concise only if there is a chain of variables x_1, \dots, x_m such that for any $i < m$, there is a literal $lif^*(x_i) - lif^*(x_{i+1}) \geq K$ in φ with a positive K . Hence the number m can be improved to $|V| + 2$ where $V = \{x \mid \text{there are } y_1, y_2 \text{ and positive } K_1, K_2 \text{ such that } lif^*(x) - lif^*(y_1) \geq K_1 \text{ and } lif^*(y_2) - lif^*(x) \geq K_2 \text{ occur in } \varphi\}$.

Moreover, when a location variable is involved in the regular $height^*$ function, the local constraint v is true and trivially satisfied by all nodes. Hence in the proof of Theorem 1, the claim $lif^*(x_i) - L \leq En + P + M + F$ can be improved to $lif^*(x_i) = 0$. As the intermittence r is trivially 1, the height bound can be improved to $(En + P + M + F + 1) \cdot ((d-1)C + 1) + D_{ht} + (m-1)D_{sub}$.

We also observed that the definitions of avl^* and rbl^* do not involve any positive constant, e.g., there is no formula $lif^*(x.dir) - lif^*(x.dir') \geq K$ with positive K . For these measure-related predicates, if they only occur positively in a $DRYAD_{dec}$ formula φ , the height bound computed in Lemma 7 can be improved, because we only need to tailor those paths with maximum number of nodes satisfying the corresponding measure function lif^* 's local constraint v . Once all of these paths are tailored, the value of lif^* is reduced by 1; moreover, these tailorings make the measure-related predicates easier to be satisfied. Hence the balancedness factor $(d-1)Cr + 1$ can be skipped and the height bound for Lemma 7 becomes $P + F + |S|$; the height bound for lif^* -related variables in Theorem 1 also can be improved to $(2En + 2P + 2F + 1) + D_{ht} + (m-1)D_{sub}$.

For CLIA synthesis, with a set of counterexamples G , there are $2^{|G|} - 1$ predicates and the height bound should be $2^{|G|} - 1$ according to Theorem 1. However, we can easily show an alternative bound which is usually better: $|G| \cdot |spec_f|$ where $|spec_f|$ is the number of distinct f -terms in φ , e.g., those terms of the form $f(v_1, \dots, v_n)$: no matter how large a decision tree T is, concretizing the $|spec_f|$ terms for each counterexample will lead to up to $|spec_f|$ leaf nodes and the whole set G will lead to up to $|G| \cdot |spec_f|$ leaf nodes in T . Let this set of leaves be S and we can tailor T to $tailor_S(T)$, which is of height up to $|G| \cdot |spec_f|$ and does not affect the evaluation of any f -term.

Performance. Our implementation leverages Z3 [33], a state-of-the-art SMT solver as the backend QF_UFLIA solver. The experiments were conducted on a server with a 40-core, 2.2GHz CPU and 128GB memory running Fedora 26.

Table 2 summarizes the experimental results on correctness verification and tree traversal fusion. For each $DRYAD_{dec}$ formula, we report the formula size,

Category	Formulae	$DRYAD_{dec}$ size Bound (Unoptimized)	$Z3$ size (KB)	Time (s) (Unoptimized)	Satisfiable?
BST_insert	nil, rec.l_pre, rec.r_pre rec.l_post, rec.r_post	≤ 48 5(11)	≤ 161	< 1 (\perp)	N
	rec.r_post_bug	48 5(11)	161	0.3 (100.5)	Y
Treap_insert	nil, rec.l_pre, rec.r_pre, rec.l_prt_le, rec.r_prt_le, rec.l_r_rtt, rec.r_l_rtt	≤ 108 7(17)	$\leq 1,696$	< 12 (\perp)	N
	rec.l_prt_le_bug,	88 7(17)	1,172	0.7 (89.8)	Y
AVL_insert (balancedness)	nil, rec.l_pre, rec.r_pre, rec.l_no_rtt, rec.l_r_rtt, rec.r_no_rtt, rec.r_l_rtt, rec.l_l_r_rtt, rec.r_r_l_rtt, rec.l_df_0, rec.r_df_0	≤ 197 7(10)	≤ 399	< 1 (< 6)	N
	rec.r_r_l_rtt_bug	197 7(10)	399	2.7 (63.2)	Y
AVL_insert (sortedness)	nil, rec.l_pre, rec.r_pre, rec.l_no_rtt, rec.r_no_rtt, rec.l_r_rtt, rec.r_l_rtt, rec.l_l_r_rtt, rec.r_r_l_rtt	≤ 134 5(11)	≤ 271	< 1 (\perp)	N
RBT_insert (balancedness)	nil, rec.l_pre, rec.r_pre, rec.l_l_blk, rec.l_r_rd, rec.l_l_rd, rec.l_all_blk, rec.r_r_blk, rec.r_l_rd, rec.r_rr_rd, rec.r_all_blk, rec.l_l_r_rd, rec.r_r_l_rd	≤ 150 7(10)	≤ 464	< 1 (< 6)	N
	l_r_rd_bug	142 7(10)	279	0.4 (9.4)	Y
RBT_insert (sortedness)	nil, rec.l_pre, rec.l_l_blk, rec.r_pre, rec.r_r_blk, rec.l_r_rd, rec.l_l_r_rd, rec.l_l_l_rd, rec.l_all_blk, rec.r_l_rd, rec.r_r_l_rd, rec.r_rr_rd, rec.r_all_blk	≤ 136 5(11)	≤ 271	< 1 (\perp)	N
InsertToLeft	nil, rec.pre, rec.post	≤ 28 7	≤ 216	< 1	N
Fusion (post_pre)	schd.l_rab, schd.rlab	4 5	84	< 1	N
	schd.l_rba, schd.rlba	4 5	84	< 1	Y
	unfusable_schd(20)	4 6	< 216	< 1	Y
Fusion (mutl_rec)	schd.lra1b2, schd.rla1b2, schd.lrb2a1, schd.rlb2a1	4 7	604	< 3	N
	unfusable_schd(20)	4 9	$< 3,304$	< 7	Y

Table 2: Performance for program verification and fusibility checking

the analytically computed height bound, the size of the encoded $Z3$ constraint in KB, the time spent by $Z3$ in seconds (\perp for timeout to 30 mins) and the satisfiability result. Bounds computed from Theorem 1 and corresponding $Z3$ running time are shown in parentheses if Bounds computed from Theorem 1 are not equal to the optimized bounds. For the program verification examples, the NPVCs generated from different basic blocks vary in their sizes, but share the same height bound. Experiments show that the height bound is critical for the performance of our decision procedure. Our bound optimization can significantly decrease the bounds, making the decision procedure scale well to solve all benchmarks. Table 3 lists the names of CLIA synthesis problems, each

Category	Formulae	DRYAD _{dec} size	Time(s)
Multiple functions	fg_fivefuncs(3), fg_sixfuncs(3), fg_sevenfuncs(3), fg_eightfuncs(3), fg_ninefuncs(3), fg_tenfunc1(3), fg_tenfunc2(3)	<279	<1
Polynomial	fg_polynomial1(3), fg_polynomial2(3), fg_polynomial3(3), fg_polynomial4(4)	<60	<1
Other CLIA	fg_max2(7), fg_VC22_a(17)	<2,227	<1
INV	ex11-new(18), ex11(17), ex14_simp(3), ex14_vars(3), formula22(1), formula25(1), formula27(1), treax1(3), trex1_vars(3), vsend(4)	<936	<1

Table 3: Performance for SyGuS benchmarks synthesis

followed by the number of formulae raised to solve it, the DRYAD_{dec} formula size and synthesis time. All queries for CLIA synthesis are solved in negligible time.

5 Related Work

It is well known that the First-Order Logic (FOL) of finite graphs is undecidable [51], and the decidability can only be obtained by restricting the logic or the class of graphs. There is a rich literature on logics over tree-like structures [7,21].

PALE [32] has been developed to verify all structures that can be expressed using graph types [21], by reducing problems to the MONA system [12]. Nonetheless, PALE and other similar techniques [11,29,57] do not reason with the data stored in the structure. Separation logic [35,45] has been a popular logic for reasoning with heap structures. Many decidable fragments have been identified. There has been significant efforts on decidable logic for structure properties of list-like structures. SLP [34] and SeLogger [6,10] are designed to check validity of the entailment problem for separation logic over pointers and lists. Iosif *et al.* [14] extend separation logic with recursive definitions to define structures of bounded tree-width, and guarantee the decidability by classical MSO reasoning.

The last decade has seen logics for reasoning about both the structure properties and data properties. The LISBQ [22] logic used in the HAVOC system is a well known decidable logic; it obtains decidability by syntactically restricting the reachability predicates and universal quantification. The CSL [3] logic is designed in a similar vein, with a different set of syntactic restrictions that allow it to express doubly-linked lists. Neither LISBQ nor CSL can handle basic tree data-structures such as binary search trees. AF^R [15] is also a decidable fragment of first-order logic with transitive closure for list-like structures. The GRIT logic [40,41] is capable to handle tree structures; its decidability is obtained by reducing the separation logic to a decidable fragment of first order logic. GRIT is decidable for reasoning local data properties, such as sortedness, but measurements of trees cannot be expressed. The STRAND logic [24,25] combines a powerful tree logic with an arbitrary data-logic. If the underlying data-logic is decidable, a fragment of STRAND is also decidable. As the first decidable logic for binary search trees, a main limitation of STRAND is it cannot express any tree measurement. In other words, AVL trees or red black trees cannot be defined.

The underlying logic in the type checker Catalyst [19] is decidable but Catalyst cannot handle measurements either. In contrast, combining term algebra and Presburger arithmetic [58,28] yields decidable theories that can model tree balancedness of red black trees, but not sortedness.

More recently, several automatic verification systems for heap-manipulating programs have been developed. Liquid Types [46,20] handle measurements by folding or unfolding the recursive definitions systematically and then treat the refined types as uninterpreted functions. As the number of unfolding or folding needed is unbounded, the system has to give up either termination or completeness. Inherited the approach from Liquid Types, LiquidHaskell [53,54,55,52] cannot guarantee termination and completeness at the same time either. Apart from DRYAD and natural proofs, by which our decidable logic is inspired, [49,50] and [4] exploit recursive definitions and proof tactics that unfold the definitions tactically. These approaches can handle arbitrary combinations of data, shape and measurement properties for trees, but give up general decidability, as mentioned in Section 1 and explained below.

Recall the `insertToLeft` example we described in Section 1. To reason about the recursively defined full-treeness and tree-size in Leon, one has to define an ad hoc abstraction function α that maps trees to the domain $(\text{Int}, \text{Boolean})$, whose first and second elements represent the tree size and full-treeness, respectively. Then Leon can decidablely verify the `insertToLeft` example only if α is *sufficiently surjective* (see Definition 7 of [49]), which is not the case. To show α is not sufficiently surjective, it suffices to find a positive integer p such that for an arbitrarily large tree t with $\alpha(t) = (i, b)$, the property $|\alpha^{-1}(i, b)| > p$ cannot be characterized by a linear arithmetic formula $M_{i,b}(c)$. Now let t be an arbitrarily large non-full tree such that $\alpha(t) = (i, \text{false})$. Notice that i , as the first part of the abstraction, represents the size of the tree t and is arbitrarily large, too. Then the term $|\alpha^{-1}(i, b)|$ essentially means the number of different non-full trees with size i . As the total number of binary trees of size i can be computed combinatorially as $\frac{(2i)!}{(i+1)! \cdot i!}$ and there is a single full tree when $i = 2^k - 1$ for some k . Hence, the property $|\alpha^{-1}(i, \text{false})| > p$ can be essentially captured by the following formula

$$M_{i,\text{false}} \equiv \frac{(2i)!}{(i+1)! \cdot i!} - \text{ite}(\exists k : i = 2^k - 1, 1, 0) > p$$

Obviously, this $M_{i,\text{false}}$ is too complicated and not equivalent to any linear arithmetic formula. Therefore, the abstract domain $(\text{Int}, \text{Boolean})$ representing size and full-treeness is not sufficiently surjective and hence cannot be reasoned by Leon in a decidable fashion.

The more recent following work [23,38] either only handle tree with bounded size in a decidable fashion or can only verify the red-black properties and the black-height of the tree, i.e., they cannot verify the functional correctness of AVL or red-black trees manipulating programs. A more recent work [56] related to Liquid Types also shows decidability for transparent formulae; but the formulae handled in our experiments are usually non-transparent.

Acknowledgments This material is based upon work supported by the National Science Foundation under Grant No. 1837023.

References

1. <https://engineering.purdue.edu/~xqiu/dryad-dec>
2. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. pp. 1–8 (2013)
3. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A Logic-Based Framework for Reasoning about Composite Data Structures, pp. 178–195. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
4. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.* pp. 1006–1036 (2012)
5. Chlipala, A.: Mostly-automated verification of low-level programs in computational separation logic. In: PLDI '11. pp. 234–245 (2011)
6. Cook, B., Haase, C., Ouaknine, J., Parkinson, M.J., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: CONCUR'11. pp. 235–249 (2011)
7. Courcelle, B.: The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation* **85**(1), 12 – 75 (1990)
8. Engelfriet, J., Maneth, S.: Output string languages of compositions of deterministic macro tree transducers. *J. Comput. Syst. Sci.* **64**(2), 350–395 (Mar 2002)
9. Goldfarb, M., Jo, Y., Kulkarni, M.: General transformations for gpu execution of tree traversals. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing). SC '13 (2013)
10. Haase, C., Ishtiaq, S., Ouaknine, J., Parkinson, M.J.: SeLogger: A tool for graph-based reasoning in separation logic. In: CAV'13. pp. 790–795 (2013)
11. Habermehl, P., Iosif, R., Vojnar, T.: Automata-based verification of programs with tree updates. *Acta Informatica* **47**(1), 1–31 (2010)
12. Heinze, T.S., Møller, A., Strocchio, F.: Type safety analysis for Dart. In: Proc. 12th Dynamic Languages Symposium (DLS) (October 2016)
13. Huang, K., Qiu, X., Tian, Q., Wang, Y.: Reconciling enumerative and symbolic search in syntax-guided synthesis (02 2018)
14. Iosif, R., Rogalewicz, A., Simáček, J.: The tree width of separation logic with recursive definitions. In: CADE-24. pp. 21–38 (2013)
15. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: CAV. pp. 756–772 (2013)
16. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: a powerful, sound, predictable, fast verifier for C and Java. In: NFM'11. pp. 41–55 (2011)
17. Jo, Y., Kulkarni, M.: Enhancing locality for recursive traversals of recursive structures. In: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications. pp. 463–482. OOPSLA '11, ACM, New York, NY, USA (2011)
18. Jo, Y., Kulkarni, M.: Automatically enhancing locality for tree traversals with traversal splicing. In: Proceedings of the 2012 ACM international conference on Object oriented programming systems languages and applications. OOPSLA '12, ACM, New York, NY, USA (2012)

19. Kaki, G., Jagannathan, S.: A relational framework for higher-order shape analysis. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 311–324. ICFP '14, ACM, New York, NY, USA (2014)
20. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 304–315. PLDI '09, ACM, New York, NY, USA (2009)
21. Klarlund, N., Schwartzbach, M.I.: Graph types. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 196–205. POPL '93, ACM, New York, NY, USA (1993)
22. Lahiri, S., Qadeer, S.: Back to the future: Revisiting precise program verification using smt solvers. In: Principles of Programming Languages (POPL '08). p. 16. Association for Computing Machinery, Inc. (January 2008)
23. Le, Q.L., Sun, J., Chin, W.N.: Satisfiability modulo heap-based programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 382–404. Springer International Publishing, Cham (2016)
24. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL'11. pp. 611–622. ACM (2011)
25. Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using STRAND. In: Yahav, E. (ed.) Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings. LNCS, vol. 6887, pp. 43–59. Springer (2011)
26. Madhusudan, P., Qiu, X., Stefanescu, A.: Recursive proofs for inductive tree data-structures. In: POPL'12. pp. 123–136. ACM (2012)
27. Maletti, A.: Compositions of extended top-down tree transducers. *Inf. Comput.* **206**(9-10), 1187–1196 (Sep 2008)
28. Manna, Z., Sipma, H.B., Zhang, T.: Verifying balanced trees. In: Artëmov, S.N., Nerode, A. (eds.) LFCS. LNCS, vol. 4514, pp. 363–378. Springer (2007)
29. McPeak, S., Necula, G.C.: Data Structure Specifications via Local Equality Axioms, pp. 476–490. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
30. Meyerovich, L.A., Bodik, R.: Fast and parallel webpage layout. In: Proceedings of the 19th International Conference on World Wide Web. pp. 711–720. WWW '10, ACM, New York, NY, USA (2010)
31. Meyerovich, L.A., Torok, M.E., Atkinson, E., Bodik, R.: Parallel schedule synthesis for attribute grammars. PPOPP '13 (2013)
32. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI'01. pp. 221–231. ACM (June 2001)
33. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
34. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic + superposition calculus = heap theorem prover. In: PLDI'11. pp. 556–566. PLDI '11 (2011)
35. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL'01. LNCS, vol. 2142, pp. 1–19. Springer (2001)
36. Pek, E., Qiu, X., Madhusudan, P.: Natural proofs for data structure manipulation in C using separation logic. In: PLDI'14. pp. 440–451. ACM (2014)
37. Petrashko, D., Lhoták, O., Odersky, M.: Miniphases: Compilation using modular and efficient tree transformations. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 201–216. PLDI 2017, ACM, New York, NY, USA (2017)

38. Pham, T., Gacek, A., Whalen, M.W.: Reasoning about algebraic data types with abstractions. CoRR **abs/1603.08769** (2016)
39. Philippaerts, P., Mühlberg, J.T., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: Industrial case studies. *Sci. Comput. Program.* **82**, 77–97 (2014)
40. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using smt. In: CAV’13 (2013)
41. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559. pp. 711–728. Springer-Verlag New York, Inc., New York, NY, USA (2014)
42. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: PLDI ’13. pp. 231–242. ACM (2013)
43. Rajbhandari, S., Kim, J., Krishnamoorthy, S., Pouchet, L.N., Rastello, F., Harrison, R.J., Sadayappan, P.: A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 40:1–40:12. SC ’16, IEEE Press, Piscataway, NJ, USA (2016)
44. Rajbhandari, S., Kim, J., Krishnamoorthy, S., Pouchet, L.N., Rastello, F., Harrison, R.J., Sadayappan, P.: On fusing recursive traversals of kd trees. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 152–162. ACM (2016)
45. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS’02. pp. 55–74. IEEE-CS (2002)
46. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 159–169. PLDI ’08, ACM, New York, NY, USA (2008)
47. Saarikivi, O., Veanes, M., Mytkowicz, T., Musuvathi, M.: Fusing effectful comprehensions. *SIGPLAN Not.* **52**(6), 17–32 (Jun 2017)
48. Sakka, L., Sundararajah, K., Kulkarni, M.: Treefuser: A framework for analyzing and fusing general recursive tree traversals. *Proc. ACM Program. Lang.* **1**(OOPSLA), 76:1–76:30 (Oct 2017)
49. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL’10. pp. 199–210 (2010)
50. Suter, P., Köksal, A.S., Kuncak, V.: Satisfiability modulo recursive programs. In: SAS’11. pp. 298–315 (2011)
51. Trakhtenbrot, B.A.: The impossibility of an algorithm for the decision problem for finite domains. *Doklady Akad. Nauk SSSR (N.S.)* **70**, 569–572 (1950)
52. Vazou, N., Bakst, A., Jhala, R.: Bounded refinement types. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. pp. 48–61. ICFP 2015, ACM, New York, NY, USA (2015)
53. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Proceedings of the 22Nd European Conference on Programming Languages and Systems. pp. 209–228. ESOP’13, Springer-Verlag, Berlin, Heidelberg (2013)
54. Vazou, N., Seidel, E.L., Jhala, R.: Liquidhaskell: experience with refinement types in the real world. In: Haskell (2014)
55. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for haskell. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 269–282. ICFP ’14, ACM, New York, NY, USA (2014)

56. Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R.G., Newton, R.R., Wadler, P., Jhala, R.: Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.* **2**(POPL), 53:1–53:31 (Dec 2017)
57. Yorsh, G., Rabinovich, A., Sagiv, M., Meyer, A., Bouajjani, A.: A Logic of Reachable Patterns in Linked Data-Structures, pp. 94–110. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
58. Zhang, T., Sipma, H.B., Manna, Z.: Decision procedures for term algebras with integer constraints. *Information and Computation* **204**(10), 1526 – 1574 (2006)