

# Recursive Proofs for Inductive Tree Data-Structures

P. Madhusudan   Xiaokang Qiu   Andrei Stefanescu

University of Illinois at Urbana-Champaign, USA

{madhu, qiu2, stefane1}@illinois.edu

## Abstract

We develop logical mechanisms and procedures to facilitate the verification of full functional properties of inductive tree data-structures using recursion that are sound, incomplete, but terminating. Our contribution rests in a new extension of first-order logic with recursive definitions called DRYAD, a syntactical restriction on pre- and post-conditions of recursive imperative programs using DRYAD, and a systematic methodology for accurately unfolding the footprint on the heap uncovered by the program that leads to finding simple recursive proofs using formula abstraction and calls to SMT solvers. We evaluate our methodology empirically and show that several complex tree data-structure algorithms can be checked against full functional specifications automatically, given pre- and post-conditions. This results in the first automatic terminating methodology for proving a wide variety of annotated algorithms on tree data-structures correct, including max-heaps, treaps, red-black trees, AVL trees, binomial heaps, and B-trees.

**Categories and Subject Descriptors** F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs: Mechanical verification; D.2.4 [Software Engineering]: Software/Program Verification: Assertion checkers

**General Terms** Algorithms, Reliability, Theory, Verification

**Keywords** heap analysis, recursive program, tree, SMT solver

## 1. Introduction

The area of program verification using theorem provers, utilizing manually provided proof annotations (pre- and post-conditions for functions, loop-invariants, etc.) has been a focus of intense research in the field of programming languages. Automatic theory solvers (SMT solvers) that handle a variety of quantifier-free theories including arithmetic, uninterpreted functions, Boolean logic, etc., serve as effective tools that automatically discharge the validity checking of many verification conditions [7].

A key area that has eluded the above paradigm of specification and verification is heap analysis: the verification of programs that dynamically allocate memory and manipulate them using pointers, maintaining structural invariants (e.g. “the nodes form a tree”), aliasing invariants, and invariants on the data stored in the locations (e.g. “the keys of a list are sorted”). The classical examples of these are the basic *data-structures* taught in undergraduate computer sci-

ence courses, and include linked lists, queues, binary search trees, max-heaps, balanced AVL trees, partially-balanced tree structures like red-black trees, etc. [10]. Object-oriented programs are rich with heap structures as well, and structures are often found in the form of records or lists of pointers pointing to other hierarchically arranged data-structures.

Dynamically allocated heaps are difficult to reason with for several reasons. First, the specification of proof annotations itself is hard, as the annotation needs to talk about several intricate properties of an unbounded heap, often requiring quantification and reachability predicates, and needs to specify aliasing as well as structural properties of the heap. Also, in experiences with manual verification, it has been observed that pre- and post-conditions get unduly complex, including large formulas that say how the frame of the heap that is *not* touched by the program remains the same across a function. Separation logic [19, 23] has emerged as a way to address this problem, mainly the frame problem mentioned above, and gives a logic that permits us to compositionally reason with the footprint touched by the program and the frame it resides in.

Most research on program logics for functional verification of heap-manipulating programs can be roughly divided into two classes:<sup>1</sup>

- **Logics for manual/semi-automatic reasoning:** The most popular of these are the class of *separation logics* [19, 23], but several others exist (see *matching logic* [24], for example). Complex structural properties of heaps are expressed using *inductive algebraic definitions*, the logic combines several other theories like arithmetic, etc., and uses a special separation operator ( $*$ ) to compositionally reason with a footprint and the frame. The analysis is either manual or semi-automatic, the latter being usually sound, incomplete, and non-terminating, and proceeds by heuristically searching for proofs using a proof system, unrolling recursive definitions arbitrarily. Typically, such tools can find simple proofs if they exist, but are unpredictable, and cannot robustly produce counter-examples.
- **Logics for completely automated reasoning:** These logics stem from the SMT (Satisfiability Modulo Theories) and automata theory literature, where the goal is to develop fast, terminating, sound and complete decision procedures, but where the logics are often constrained heavily on expressivity in order to reach these goals. Examples include several logics that extend first-order logic with reachability, the logics LISBQ [14] and CSL [6], and the logic STRAND<sub>dec</sub> [16] that combines tree theories with integer theories. The problem with these logics, in general, is that they are often not sufficiently expressive to state complex properties of the heap (e.g. the balancedness of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.  
Copyright © 2012 ACM 978-1-4503-1083-3/12/01...\$10.00

<sup>1</sup> We do not discuss abstraction-based approaches such as shape analysis here as such approaches are geared towards less complex specifications, and often are completely automatic, not even requiring proof annotations such as loop invariants; see Section 6.

an AVL tree, or that the set of keys stored in a heap do not change across a program).

We prefer an approach that combines the two methodologies above. We propose a strategy that (a) identifies a class of *simple and natural* proofs for proving verification conditions for heap-based programs, founded on how people prove these conditions manually, and (b) builds terminating procedures that efficiently and thoroughly search this class of proofs. This results in a sound, incomplete, but terminating procedure that finds natural proofs automatically and efficiently. Many correct programs have simple proofs of correctness, and a terminating procedure that searches for these simple proofs efficiently can be a very useful tool in program verification. Incompleteness is, of course, a necessary trade-off to keep the logics expressive while having a terminating procedure, and a terminating automatic procedure is useful as it does not need manual help. Furthermore, as we shall see in this paper, such decision algorithms are particularly desirable when they can be made to work very efficiently, especially using the fast-growing class of efficient SMT solvers for quantifier-free theories.

The idea of searching for only simple and natural proofs is not new; after all, *type systems* that prove properties of programs are essentially simple (and often scalable) proof mechanisms. The class of simple and natural proofs that we identify in this paper is, however, quite different from those found by type systems.

In this paper, we develop logical mechanisms to identify a simple class of proofs based on a deterministic proof tactic that (a) unfolds recursive terms *precisely* across the footprint, (b) uses *formula abstractions* (that replace recursively defined terms with *uninterpreted terms*) to restate the verification condition in a quantifier-free decidable theory, and (c) checks the resulting formula using an automatic decision procedure. We deploy this technique for the specialized domain of *imperative programs manipulating tree data-structures*, developing an extension of first-order logic with recursion on trees called DRYAD to state properties of programs, and building procedures based on precise unfoldings and formula abstractions.

### Motivating formula abstractions

When reasoning with formulas that have recursively defined terms, which can be unrolled forever, a key idea is to use *formula abstraction* that makes the terms *uninterpreted*. Intuitively, the idea is to replace recursively defined predicates, sets, etc. by uninterpreted Boolean values, uninterpreted sets, etc.

The idea of formula abstraction is extremely natural, and utilized very often in manual proofs. For instance, let us consider a binary search tree (BST) search routine searching for a key  $k$  on the root node  $x$ . The verification condition of a path of this program, typically, would require checking:

$$(bst(x) \wedge k \in keys(x) \wedge k < x.key) \Rightarrow (k \in keys(x.left))$$

where  $bst()$  is a recursive predicate defined on trees that identifies binary search trees, and  $keys()$  is a recursively defined set that collects the multiset of keys under a node. Unrolling the definition of  $keys()$  and  $bst()$  gives the following formula (below,  $i < S$  means that  $i$  is less than every element in  $S$ ):

$$bst(x.left) \wedge bst(x.right) \wedge keys(x.left) < x.key \wedge keys(x.right) > x.key \wedge k \in (keys(x.left) \cup keys(x.right) \cup \{x.key\}) \wedge k < x.key \Rightarrow (k \in keys(x.left))$$

Now, while the above formula is quite complex, involving recursive definitions that can be unrolled *ad infinitum*, we can prove its validity soundly by viewing  $bst()$  and  $keys()$  as *uninterpreted functions* that map locations to Booleans and sets, respectively. Doing this gives (modulo some renaming and modulo theory of equality):  $(b_1 \wedge b_2 \wedge K_1 \leq xkey \wedge K_2 > xkey \wedge k \in (K_1 \cup K_2 \cup \{xkey\}) \wedge k < xkey) \Rightarrow (k \in K_1)$

Note that the above formula is a quantifier-free formula over integers and multisets of integers, and furthermore is valid (since  $k < xkey$  and  $K_2 > xkey$ ,  $k$  must be in  $K_1$ ). Validity of quantifier-free formulas over sets/multisets of integers with addition is *decidable* (they can be translated to quantifier-free formulas over integers and uninterpreted functions), and can be solved using SMT solvers efficiently. Consequently, we can prove that the verification condition is valid, completely automatically. Note that formula abstraction is *sound* but *incomplete*.

This idea has been explored in the literature. For example, Suter et al. [25] have proposed abstraction schemes for algebraic data-types that soundly (but incompletely) transform logical validity into much simpler decidable problems using formula abstractions, and developed mechanisms for proving *functional programs* correct.

### Reasoning with DRYAD

In order to build the procedures for reasoning with programs manipulating trees, using precise unfolding and abstraction, we develop a new recursive extension of first-order logic, called DRYAD that allows stating complex properties of heaps without recourse to explicit quantification. DRYAD combines quantifier-free first-order logic with recursive definitions, and these recursive definitions, themselves expressed in DRYAD, can capture several interesting properties of trees, including their height, the multiset of keys stored in them, whether they correspond to a binary search tree (or an AVL tree), etc.

The main technical contribution of this paper is to show how a Hoare-triple corresponding to a basic path in a recursive imperative program (we disallow while-loops and demand all recursion be through recursive function calls) with proof annotations written in DRYAD, can be expressed as a pair consisting of a finite footprint and a DRYAD formula. The finite footprint is a *symbolic heap* that captures the heap explored by the basic block of the program precisely. The construction of this footprint and formula calls for a careful handling of the mutating footprint defined by a recursive imperative program, calls for a disciplined approach to unrolling recursion, and involves capturing aliasing and separation by exploiting the fact that the manipulated structures are trees. In particular, the procedure keeps track of locations in the footprint corresponding to trees and precisely computes the value of recursive terms on these. Furthermore, the verification condition is accurately described by unfolding the pre-condition so that it is expressed purely on the *frontier* of the footprint, so as to enable effective use of the formula abstraction mechanism. In order to be accurate, we place several key restrictions on the logical syntax of pre- and post-conditions expressed for functions.

We then consider the problem of solving the validity problem for the verification condition expressed as a footprint and a DRYAD formula. We first show a small syntactic fragment of verification conditions that is entirely decidable (without formula abstraction) by a reduction to the decidable logic STRAND<sub>dec</sub> [16]. Although this fragment of DRYAD is powerful enough to express certain restricted structural properties of simple tree data-structures like binary search trees, max-heaps, and treaps, it is completely inadequate in verifying more complex properties of the above structures as well as properties of more complex tree data-structures such as red-black trees, binomial heaps, etc.

We turn next to abstraction schemes for DRYAD, and show how to abstract DRYAD formulas into quantifier-free theories of sets/multisets of integers; the latter can then be translated into formulas in the standard quantifier-free theory of integers with uninterpreted functions. The final formula's validity can be proved using standard SMT solvers, and its validity implies the validity of the DRYAD formula.

$dir \in Dir$	$i^* : Loc \rightarrow Int$	$x \in Loc$ Variables	$S \in \mathcal{S}(Int)$ Variables
$f \in DF$	$si^* : Loc \rightarrow \mathcal{S}(Int)$	$j \in Int$ Variables	$MS \in \mathcal{MS}(Int)$ Variables
$p^* : Loc \rightarrow \{\text{true}, \text{false}\}$	$msi^* : Loc \rightarrow \mathcal{MS}(Int)$	$q \in Boolean$ Variables	$c : Int$ Constant
$Loc$ Term: $lt, lt_1, lt_2, \dots ::= x \mid \text{nil} \mid lt.dir$ $Int$ Term: $it, it_1, it_2, \dots ::= c \mid j \mid lt.f \mid i^*(lt) \mid it_1 + it_2 \mid it_1 - it_2 \mid \text{ite}(\varphi, it_1, it_2)$ $\mathcal{S}(Int)$ Term: $sit, sit_1, sit_2, \dots ::= \emptyset \mid S \mid \{it\} \mid si^*(lt) \mid sit_1 \cup sit_2 \mid sit_1 \cap sit_2 \mid sit_1 \setminus sit_2 \mid \text{ite}(\varphi, sit_1, sit_2)$ $\mathcal{MS}(Int)$ Term: $msit, msit_1, msit_2, \dots ::= \emptyset_m \mid MS \mid \{it\}_m \mid msi^*(lt) \mid msit_1 \cup msit_2 \mid msit_1 \cap msit_2 \mid msit_1 \setminus msit_2 \mid \text{ite}(\varphi, msit_1, msit_2)$			
$Formula: \varphi, \varphi_1, \varphi_2, \dots ::= \text{true} \mid q \mid p^*(lt) \mid lt_1 = lt_2 \mid it_1 \leq it_2 \mid sit_1 \subseteq sit_2 \mid msit_1 \subseteq msit_2 \mid sit_1 \leq sit_2 \mid msi_1 \leq msi_2 \mid it \in sit \mid it \in msit \mid \neg\varphi \mid \varphi_1 \vee \varphi_2$			
$Recursively\text{-defined integer} : i^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil}, i_{base}, i_{ind})$ $Recursively\text{-defined set-of-integers} : si^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil}, si_{base}, si_{ind})$ $Recursively\text{-defined multiset-of-integers} : msi^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil}, msi_{base}, msi_{ind})$ $Recursively\text{-defined predicate} : p^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil}, p_{base}, p_{ind})$			

**Figure 1.** Syntax of DRYAD

Finally, we evaluate our logical mechanisms and procedures, by writing several tree data-structure algorithms using pure recursive imperative programs, annotating them using DRYAD in order to state their complete functional correctness, derive the verification conditions expressed as footprints and DRYAD formulae, and prove them valid using the formula abstraction scheme. Much to our surprise, all verification conditions in all the programs were proved valid automatically and efficiently by our procedure.

We have verified full functional correctness of data-structures ranging from sorted linked lists, binary search trees, max-heaps, treaps (which are binary search trees on the first key and max-heaps on the second), AVL trees and red-black trees (semi-balanced search trees), B-trees, and binomial heaps. This set of benchmarks is an almost exhaustive list of algorithms on tree-based data-structures covered in a first undergraduate course on data-structures [10]. To the best of our knowledge, the work presented here is the first methodology that can prove such a wide variety of algorithms on tree data-structures written in an imperative language fully functionally correct using a sound and terminating procedure.

## 2. The DRYAD Logic for Heaps

The recursive logic over trees, DRYAD, is essentially a *quantifier-free* first-order logic over heaps augmented with recursive definitions of various types (e.g., integers, sets/multisets of integers, etc.) defined for locations that have a tree under them. While first-order logic gives the necessary power to talk precisely about locations that are near neighbors, the recursive definitions allow expressing properties that require quantifiers, including reachability, collecting the set/multiset of keys in a tree, and defining natural metrics, like the height of a tree, that are typically useful in defining properties of trees.

Given a finite set of *directions*  $Dir$ , let us define *Dir*-trees as finite trees where every location has either  $|Dir|$  children, or is the  $\text{nil}$  location, which has no children (we assume there is a single  $\text{nil}$  location). Binary trees have two directions:  $Dir = \{l, r\}$ .

The logic DRYAD is parameterized by a finite set of *directions*  $Dir$  and also by a finite set of *data-fields*  $DF$ . Let us fix these sets.

Let  $Bool = \{\text{true}, \text{false}\}$  stand for the set of Boolean values,  $Int$  stand for the set of integers and  $Loc$  stand for the universe of locations. For any set  $A$ , let  $\mathcal{S}(A)$  denote the set of subsets of  $A$ , and let  $\mathcal{MS}(A)$  denote the set of all multisets with elements in  $A$ .

The DRYAD logic allows four kinds of recursively defined notions for a location that is the root of a *Dir*-tree: recursively

defined integer functions ( $Loc \rightarrow Int$ ), recursively defined set-of-keys/integers functions ( $Loc \rightarrow \mathcal{S}(Int)$ ), recursively defined multiset-of-keys/integers functions ( $Loc \rightarrow \mathcal{MS}(Int)$ ), and recursively defined Boolean predicates ( $Loc \rightarrow Bool$ ). Let us fix disjoint sets of countable names for such functions. We will refer to these recursive functions as recursively defined integers, recursively defined sets/multisets of integers, and recursively defined predicates, respectively. Typical examples of these include the height of a tree or the height of black-nodes in the tree rooted at a node (recursively defined integers), the set/multiset of keys stored at a particular data-field under nodes (recursively defined set/multiset of integers), and the property that the tree rooted at a node is a binary search tree or a balanced tree (recursively defined predicates).

A DRYAD formula consists of a pair  $(Def, \varphi)$ , where  $Def$  is a set of recursive definitions and  $\varphi$  is a formula. The syntax of DRYAD logic is given in Figure 1, where the syntax of the formulas is followed by the syntax for recursive definitions. We require that every recursive function/predicate used in the formula  $\varphi$  has a unique definition in  $Def$ . The figure does not define the syntax of the base and inductive formulas in recursive definitions (e.g.  $i_{base}$ ,  $i_{ind}$ , etc.); we give that in the text below.

Location terms are formed using pointer fields from location variables, and include a special location called  $\text{nil}$ . Integer terms are obtained from integer constants, data-fields of locations, and from recursively defined integers, and combined using basic arithmetic operations of addition and subtraction and conditionals ( $\text{ite}$  stands for if-then-else terms that evaluate to the second argument if the first argument evaluates to  $\text{true}$  and evaluate to the third argument otherwise).

Terms that evaluate to a set/multiset of integers are obtained from recursively defined sets/multisets of integers corresponding to a location term, and are combined using set/multiset operations as well as conditional choices. Formulas are obtained by Boolean combinations of Boolean variables, recursively defined predicates on a location term, and using various relations between set and multiset terms. The relations on sets and multisets include the subset relation as well as the relation  $\leq$  which is interpreted as follows: for two sets (or multisets) of integers  $S_1$  and  $S_2$ ,  $S_1 \leq S_2$  holds whenever for every  $i \in S_1$ ,  $j \in S_2$ ,  $i \leq j$ .

The recursively defined functions (or predicates) are defined using the syntax:  $f^*(x) = \text{ite}(x = \text{nil}, f_{base}, f_{ind})$ , where  $f_{base}$  and  $f_{ind}$  are themselves terms (or formulas) that stand for what  $f$  evaluates to when  $x = \text{nil}$  (the base-case) and when  $x \neq \text{nil}$  (the

inductive step), respectively. There are several restrictions on these terms/formulas:

- $f_{base}$  has no free variables and hence evaluates to a fixed value (for integers, it is a fixed integer; for sets/multisets of integers, it is a fixed set; for Boolean predicates, it evaluates to `true` or `false`).
- $f_{ind}$  only has  $x$  as a free variable. Furthermore, the location terms in it can only be  $x$  and  $x.dir$  (further dereferences are disallowed). Moreover, integer terms  $x.dir.f$  are disallowed.

Intuitively, the above conditions demand that when  $x$  is `nil`, the function evaluates to a constant of the appropriate type, and when  $x \neq \text{nil}$ , it evaluates to a function that is defined recursively using properties of the location  $x$ , which may include properties of the children of  $x$ , and these properties may in turn involve other recursively defined functions.

We assume that the inductive definitions are not *circular*. Formally, let  $Def$  be a set of definitions and consider a recursive definition of a function  $f^*$  in  $Def$ . Define the sequence  $\psi_0, \psi_1, \dots$  as follows. Set  $\psi_0 = f^*(x)$ . Obtain  $\psi_{i+1}$  by replacing every occurrence of  $g^*(x)$  in  $\psi_i$  by  $g_{ind}(x)$ , where  $g$  is any recursively defined function in  $Def$ . We require that this sequence eventually stabilizes (i.e. there is a  $k$  such that  $\psi_k = \psi_{k+1}$ ). Intuitively, we require that the definition of  $f^*(x)$  be rewritable into a formula that does not refer to a recursive definition of  $x$  (by getting rewritten to properties of its descendants). We require that every definition in  $Def$  have the above property.

**Example: Red Black Trees** Red black trees are semi-balanced binary search trees with nodes colored red and black, with all the leaves colored black, satisfying the condition that the left and right children of a red node are black, and the condition that the number of black nodes on paths from the root to any leaf is the same. This ensures that the longest path from root to a leaf is at most twice the shortest path from root to a leaf, making the tree roughly balanced.

We have two directions  $Dir = \{l, r\}$ , and two data fields,  $key$ , and  $color$ . We model the color of nodes using an integer data-field  $color$ , which can be 0 (black) or 1 (red). We define four recursive functions/predicates: a predicate  $black^*(x)$  that checks whether the root of the tree under  $x$  is colored black (this is defined as a recursive predicate for technical reasons), the black height of a tree,  $bh^*(x)$ , the multiset of keys stored in a tree,  $keys^*(x)$ , and a recursive predicate that identifies red-black trees,  $rbt^*(x)$ .

$$\begin{aligned}
black^*(x) &\stackrel{def}{=} \text{ite}(x = \text{nil}, \text{true}, x.color = 0) \\
bh^*(x) &\stackrel{def}{=} \text{ite}(x = \text{nil}, 1, \text{ite}(x.color = 0, 1, 0) + \\
&\quad \text{ite}(bh^*(x.l) \geq bh^*(x.r), bh^*(x.l), bh^*(x.r))) \\
keys^*(x) &\stackrel{def}{=} \text{ite}(x = \text{nil}, \emptyset, \{x.key\} \cup keys^*(x.l) \cup keys^*(x.r)) \\
rbt^*(x) &\stackrel{def}{=} \text{ite}(x = \text{nil}, \text{true}, \\
&\quad rbt^*(x.l) \wedge rbt^*(x.r) \wedge \\
&\quad keys^*(x.l) \leq \{x.key\} \wedge \{x.key\} \leq keys^*(x.r) \wedge \\
&\quad (x.color = 1 \rightarrow (black^*(x.l) \wedge black^*(x.r))) \wedge \\
&\quad bh^*(x.l) = bh^*(x.r))
\end{aligned}$$

The  $bh^*(t)$  function definition says that the black height of a tree is 1 for a `nil` node (`nil` nodes are assumed to be black), and, otherwise, the maximum of the black heights of the left and right subtree if the node  $x$  is *red*, and the maximum of the black heights of the left and right subtree plus one, if  $x$  is *black*. The  $keys^*(t)$  function says that the multiset of keys stored in a tree is  $\emptyset$  for a `nil`-node, and the union of the key stored in the node, and the keys of the left and right subtrees. Finally, the  $rbt^*(t)$  holds if: (1) the left and right subtrees are valid red black trees; (2) the keys of the left subtree are no greater than the key in the node, and the keys of the right subtree are no less than the key in the node; (3) if the node is

*red*, both its children are *black*; and (4) the black heights of the left and the right subtrees are equal.

We can also express, in our logic, various properties of red black trees, by including the above definitions and a formula like:

$$(rbt^*(t) \wedge \neg black^*(t) \wedge t.key = 20) \rightarrow 10 \notin keys^*(t.r)$$

and using the procedures outlined in this paper, check the validity of the above statement.

## Semantics

The DRYAD logic is interpreted on (concrete) heaps. Let us fix a finite set of *program variables*  $PV$ . Concrete heaps are defined as follows ( $f : A \rightarrow B$  denotes a partial function from  $A$  to  $B$ ):

**Definition 2.1.** A concrete heap over a set of directions  $Dir$ , a set of data-fields  $DF$ , and a set of program variables  $PV$  is a tuple

$$(N, nil, pf, df, pv)$$

where:

- $N$  is a finite or infinite set of locations;
- $nil \in N$  is a special location representing the null pointer;
- $pf : (N \setminus \{nil\}) \times Dir \rightarrow N$  is a function defining the direction fields;
- $df : (N \setminus \{nil\}) \times DF \rightarrow \mathbb{Z}$  is a function defining the data-fields;
- $pv : PV \rightarrow N \cup \mathbb{Z}$  is a partial function mapping program variables to locations or integers, depending on the type of the variable.  $\square$

A concrete heap consists of a finite/infinite set of locations, with a pointer-field function  $pf$  that maps locations to locations for each direction  $dir \in Dir$ , a data-field function  $df$  mapping locations to integers for each data-field  $DF$ , along with a unique constant location representing `nil` that has no data-fields or pointer-fields from it. Moreover, the function  $pv$  is a partial function that maps program variables to locations and integers.

A DRYAD formula with free variables  $F$  is interpreted by interpreting the program variables in  $F$  according to the function  $pv$  and the other variables being given an interpretation (hence, for validity, these other variables are universally quantified, and for satisfiability, they are existentially quantified).

Each term evaluates to either a normal value of the corresponding type, or to `undef`. A location term is evaluated by dereferencing pointers in the heap. If a dereference is undefined, the term evaluates to `undef`. The set of locations that are roots of  $Dir$ -trees are special in that they are the only ones over which recursive definitions are properly defined. A term of the form  $i^*(lt)$ ,  $si^*(lt)$  or  $msi^*(lt)$  will evaluate to `undef` if  $lt$  evaluates to `undef` or is not a root of a tree in the heap; otherwise it will be evaluated inductively using its recursive definition. Other aspects of the logic are interpreted with the usual semantics of first-order logic, unless they contain some subterm evaluating to `undef`, in which case they also evaluate to `undef`.

Each DRYAD formula evaluates to either `true` or `false`. To evaluate a formula  $\varphi$ , we first convert  $\varphi$  to its *negation normal form* (NNF), and evaluate each atomic formula of the form  $p^*(lt)$  first. If  $lt$  is not undefined,  $p^*(lt)$  will be evaluated inductively using the recursive definition of  $p^*$ ; if  $lt$  evaluates to `undef`,  $p^*(lt)$  will evaluate to `false` if  $p^*(lt)$  appears positively, and will evaluate to `true` otherwise. Intuitively, undefined recursive predicates cannot help in making the formula true over a model. Similarly, atomic formulas involving terms that evaluate to `undef` are set to false or true depending on whether the atomic formula occurs within an even or odd number of negations, respectively. All other relations between integers, sets, and multisets are interpreted in the natural way, and we skip defining their semantics.

We assume that the DRYAD formulas always include a recursively defined predicate *tree* that is defined as:

$$tree^*(x) \stackrel{def}{=} (x = \text{nil}, \text{true}, \text{true})$$

Note that since recursively defined predicates can hold only on trees and since the above formula vacuously holds on any tree,  $tree^*(x)$  holds iff  $x$  is a root of a *Dir*-tree.

### Programs and basic blocks

We consider imperative programs manipulating heap structures and the data contained in the heap. In this paper, we assume that programs do not contain *while* loops and all recursion is captured using recursive function calls. Consequently, proof annotations only involve pre- and post-conditions of functions, and there are no loop-invariants.

The imperative programs we analyze will consist of integer operations, heap operations, conditionals and recursion. In order to verify programs with appropriate proof annotations, we need to verify linear blocks of code, called *basic blocks*, which do not have conditionals (conditionals are replaced with *assume* statements). Basic blocks always start from the *beginning* of a function and either end at an assertion in the program (checking an intermediate assertion), or end at a function call to check whether the pre-condition to calling the function holds, or ends at the end of the program in order to check whether the post-condition holds. Basic blocks can involve recursive and non-recursive function calls.

We define basic blocks using the following grammar, parameterized by a set of directions *Dir* and a set of data-fields *DF*:

$$\begin{aligned} bb & :- bb'; \text{return } u; \mid bb'; \text{return } j; \\ bb' & :- bb'; bb' \mid u := v \mid u := \text{nil} \mid u := v.dir \mid u.dir := v \mid \\ & \quad j := u.f \mid u.f := j \mid u := \text{new} \mid j := aexpr \mid \\ & \quad \text{assume } (bexpr) \mid u := f(v, z_1, \dots, z_n) \mid j := g(v, z_1, \dots, z_n) \\ aexpr & :- j \mid aexpr + aexpr \mid aexpr - aexpr \\ bexpr & :- u = v \mid u = \text{nil} \mid aexpr \leq aexpr \mid \neg bexpr \mid bexpr \vee bexpr \end{aligned}$$

Since we deal with tree data-structure manipulating programs, which often involve functions that take as input a tree and return a tree, we make certain crucial assumptions. One crucial restriction we assume for the technical exposition is that all functions take in at most *one* location parameter as input (the rest have to be integers). Basic blocks hence have function calls of the form  $f(v, z_1, \dots, z_n)$ , where  $v$  is the only location parameter. This restriction greatly simplifies the proofs as it is much easier to track one tree. We can relax this assumption, but when several trees are passed as parameters, our decision procedures will implicitly assume a pre-condition that the trees are all disjoint. This is crucial; our decision procedures cannot track trees that “collide”; they track only equal trees and disjoint trees. This turns out to be a natural property of most data-structure manipulating programs.

### Pre- and Post-conditions of functions

We place stringent restrictions on annotations pre- and post-functions that we allow in our framework, and these are important for our technique and is the price we pay for automation. Recall that we allow only two kinds of functions, one returning a location  $f(v, z_1, \dots, z_n)$  and one returning an integer  $g(v, z_1, \dots, z_n)$  ( $v$  is a location parameter,  $z_1, \dots, z_n$  are integer parameters). We require that  $v$  is the root of a *Dir*-tree at the point when the function is called, and this is an implicit pre-condition of the function called.

Each function is annotated with its pre- and post-conditions using *annotating formulas*. Annotating terms and formulas are DRYAD terms and formulas that do not refer to any child or any data field, do not allow any equality between locations and do not allow *ite*-expressions. We denote the pre-condition as a *pre-annotating formula*  $\psi(v, z_1, \dots, z_n)$ .

The post-condition annotation is more complex, as it can talk about properties of the heap at the pre-state as well as the post-state. We allow combining terms and formulas obtained from the pre-heap and the post-heap to express the post-condition. Terms and formulas over the post-heap are obtained using DRYAD annotating terms and formulas that are allowed to refer to a variable  $old\_v$  which points to the location  $v$  pointed to in the pre-heap. These terms and formulas can also refer to the variable  $ret\_loc$  or  $ret\_int$  to refer to the location or integer being returned. Terms and formulas over the pre-heap are obtained using DRYAD annotating terms and formulas referring to  $old\_v$  and  $old\_z_i$ 's, except that all recursive definitions are renamed to have the prefix *old*\_. Then a *post-annotating formula* combines terms and formulas expressed over the pre-heap and the post-heap (using the standard operations).

For a function  $f(v, z_1, \dots, z_n)$  that returns a location, we assume that the returned location always has a *Dir*-tree under it (and this is implicitly assumed to be part of the post-condition). The post-condition for  $f$  is either of the form

$$\text{havoc}(old\_v) \wedge \psi(old\_v, old\_z_1, \dots, old\_z_n, ret\_loc)$$

or of the form

$$old\_v \# ret\_loc \wedge \psi(old\_v, old\_z_1, \dots, old\_z_n, ret\_loc)$$

where  $\psi$  is a post-annotating formula. In the first kind,  $\text{havoc}(old\_v)$  means that the function guarantees nothing about the location pointed to in the pre-state by the input parameter  $v$  (and nothing about the locations accessible from that location) and hence the caller of  $f$  cannot assume anything about the location it passed to  $f$  after the call returns. In that case, we restrict  $\psi$  from referring to  $r^*(old\_v)$ , where  $r^*$  is a recursive predicate/function on the post-heap. In the latter kind  $old\_v \# ret\_loc$  means that  $f$ , at the point of return, assures that the location passed as parameter  $v$  now points to a *Dir*-tree and this tree is disjoint from the tree rooted at  $ret\_loc$ .

In either case, the formula  $\psi$  can relate complex properties of the returned location and the input parameter, including recursive definitions on the old parameter and the new ones. For example, a post-condition of the form  $\text{havoc}(old\_v) \wedge keys^*(old\_v) = keys^*(ret\_loc)$  says that the keys under the returned location are precisely the same as the keys under the location passed to the function.

For a function  $g$  returning an integer, the post-condition is of the form

$$\text{tree}(old\_v) \wedge \psi(old\_v, old\_z_1, \dots, old\_z_n, ret\_int)$$

or of the form

$$\text{havoc}(old\_v) \wedge \psi(old\_v, old\_z_1, \dots, old\_z_n, ret\_int)$$

The former says that the location passed as input continues to point to a tree, while the latter says that no property is assured about the location passed as input (same restriction on  $\psi$  applies).

The above restriction that the input tree and the returned tree either point to completely disjoint trees or that the input pointer (and nodes accessible from it) are entirely havoc-ed and the returned node is some tree are the only separation and aliasing properties that the post-condition can assert. Our logical mechanism is incapable, for example, of saying the the returned node is a reachable node from the location passed to the function. We have carefully chosen such restrictions in order to simplify tracking tree-ness and separation in the footprint. In practice, most data-structure algorithms fall into these categories (for example, an insert routine would havoc the input tree and return a new tree whose keys are related to the keys of the input tree, while a tree-copying program will return a tree disjoint from the input tree).

### 3. Describing the Verification Condition in DRYAD

We now present the main technical contribution of this paper: given a set of recursive definitions, and a Hoare-triple  $(\varphi_{pre}, bb, \varphi_{post})$ , where  $bb$  is a basic block, we show how to systematically define the verification condition corresponding to it. Note that since we do not have while-loops, basic blocks always start at the beginning of a function and go either till the end of the function (spanning calls to other functions) or go up to a function call (in order to check if the pre-condition for that call holds). In the former case, the post-condition is a post-condition annotation. In the latter case, we need another form:

$$\text{tree}(y) \wedge \psi(\bar{x})$$

where  $\bar{x}$  is a subset of program variables. The pre-condition of the called function implicitly assumes that the input location is a tree (which is expressed using  $\text{tree}(y)$  above), and the pre-condition itself is adapted (after substituting formal parameters with actual terms passed to the function) and written as the formula  $\psi$ .

This verification condition is expressed as a combination of (a) quantifier-free formulas that define properties of the footprint the basic block uncovers on the heap, combined with (b) recursive formulas expressed only on the frontier of the footprint.

This verification condition is formed by unrolling recursive definitions appropriately as the basic block increases its footprint so that recursive properties are translated to properties of the frontier. This allows us to write the (strongest) post-condition of  $\varphi_{pre}$  on precisely the same nodes as  $\varphi_{post}$  refers to, which then allows us to apply *formula abstractions* to prove the verification condition. Also, recursive calls to functions that process the data-structure recursively are naturally called on the frontier of the footprint, which allows us to summarize the call to the function on the frontier.

We define the verification condition using two steps. In the first step, we inductively define a *footprint structure*, composed of a *symbolic heap* and a DRYAD formula, which captures the state of the program that results when the basic block executes from a configuration satisfying the pre-condition. We then incorporate the post-condition and derive the verification condition.

A symbolic heap is defined as follows:

**Definition 3.1.** A symbolic heap over a set of directions  $Dir$ , a set of data-fields  $DF$ , and a set of program variables  $PV$  is a tuple

$$(C, S, I, c_{nil}, pf, df, pv)$$

where:

- $C$  is a finite set of concrete nodes;
- $S$  is a finite set of symbolic tree nodes with  $C \cap S = \emptyset$ ;
- $I$  is a set of integer variables;
- $c_{nil} \in C$  is a special concrete node representing  $nil$ ;
- $pf : (C \setminus \{c_{nil}\}) \times Dir \rightarrow C \cup S$  is a partial function mapping every pair of a concrete node and a direction to nodes (concrete or symbolic);
- $df : (C \setminus \{c_{nil}\}) \times DF \rightarrow I$  is a partial function mapping concrete nodes and data-fields pairs to integer variables;
- $pv : PV \rightarrow C \cup S \cup I$  is a partial function mapping program variables to nodes or integer variables (location variables are mapped to  $C \cup S$  and integer variables to  $I$ ).  $\square$

Intuitively, a symbolic heap  $(C, S, I, c_{nil}, pf, df, pv)$  has two finite sets of nodes: concrete nodes  $C$  and symbolic tree nodes  $S$ , with the understanding that each  $s \in S$  stands for a node that may have an arbitrary *Dir*-tree under it, and furthermore the separation constraint that for any two symbolic tree nodes  $s, s' \in S$ , the trees under it would not intersect with each other, nor with the nodes in  $C$ . The tree under a symbolic node is not represented in the

symbolic heap at all. One of the concrete nodes ( $c_{nil}$ ) represents the  $nil$  location.

The function  $pf$  captures the pointer-field  $dir$  in the heap that is within the footprint, and maps the set of concrete nodes to concrete and symbolic nodes. The pointer fields of symbolic nodes are not modeled, as they are part of the tree below the node that is not represented in the footprint. The functions  $df$  and  $pv$  capture the data-fields (mapping to integer variables) and program variables restricted to the nodes in the symbolic heap.

A symbolic heap hence represents a (typically infinite) set of concrete heaps, namely those in which it can be embedded. We define this formally using the notion of *correspondence* that captures when a concrete heap is represented by a symbolic heap.

**Definition 3.2.** Let  $SH = (C, S, I, c_{nil}, pf, df, pv)$  be a symbolic heap and let  $CH = (N, nil, pf', df', pv')$  be a concrete heap. Then  $CH$  is said to correspond to  $SH$  if there are two function  $h : C \cup S \rightarrow N$  such that the following conditions hold:

- $h(c_{nil}) = nil$ ;
- for any  $n, n' \in C$ , if  $n \neq n'$ , then  $h(n) \neq h(n')$ ;
- for any two nodes  $n \in C \setminus \{c_{nil}\}$ ,  $n' \in C \cup S$ , and for any  $dir \in Dir$ , if  $pf(n, dir) = n'$ , then  $pf'(h(n), dir) = h(n')$ ;
- for any  $s \in S$ ,  $h(s)$  is the root of a *Dir*-tree in  $CH$ , and there is no concrete node  $c \in C \setminus \{c_{nil}\}$  such that  $h(c)$  belongs to this tree;
- for any  $s, s' \in S$ ,  $s \neq s'$ , the *Dir*-trees rooted at  $h(s)$  and  $h(s')$  (in  $CH$ ) are disjoint except for the  $nil$  node;
- for any location variable  $v \in PV$ , if  $pv(v)$  is defined, then  $pv'(v) = h(pv(v))$ ;  $\square$

Intuitively,  $h$  above defines a restricted kind of homomorphism between the nodes of the symbolic heap  $SH$  and a portion of the concrete heap  $CH$ . Distinct concrete non- $nil$  nodes are required to map to distinct locations in the concrete heap. Symbolic nodes are required to map to trees that are disjoint (save the  $nil$  location); they can map to the  $nil$  location as well. The trees rooted at locations corresponding to symbolic nodes must be disjoint from the locations corresponding to concrete nodes. Note that there is no requirement on the integer variables  $I$  and the map  $pv'$  on integer variables and the maps  $df$  and  $df'$ . Note also that for a concrete node in the symbolic heap  $n$ , the fields defined from  $n$  in the symbolic heap must occur in the concrete heap as well from the corresponding location  $h(n)$ ; however, the fields not defined for  $n$  may or may not be defined on  $h(n)$ .

A *footprint* is a pair  $(SH; \varphi)$  where  $SH$  is a symbolic heap and  $\varphi$  is a DRYAD formula. The semantics of such a footprint is that it represents all concrete heaps that both correspond to  $SH$  and satisfy  $\varphi$ .

#### Tree-ness of nodes in symbolic heaps

The key property of a symbolic heap is that we can determine that certain nodes have *Dir*-trees rooted under them (i.e. in any concrete heap corresponding to the symbolic heap, the corresponding locations will have a *Dir*-tree under them).

For a symbolic heap  $SH = (C, S, I, c_{nil}, pf, df, pv)$ , let the set of *graph nodes* of  $SH$  be the smallest set of nodes  $V \subseteq C \cup S$  such that:

- $c_{nil} \in V$  and  $S \subseteq V$
- For any node  $n \in C$ , if for every  $dir \in Dir$ ,  $pf(n, dir)$  is defined and belongs to  $V$ , then  $n \in V$ .

Now define  $Graph(SH)$  to be the directed graph  $(V, E)$ , where  $V$  is as above, and  $E$  is the set of edges  $(u, v)$  such that  $pf(u, dir) = v$  for some  $dir \in Dir$ . Note that, by definition, there are no edges out of  $u$  if  $u \in S$ , as symbolic nodes do not have outgoing fields.

We say that a node  $u$  in  $V$  is the root of a tree in  $Graph(SH)$  if the set of all nodes reachable from  $u$  forms a tree (in the usual graph-theoretic sense).

The following claim follows and is the crux of using the symbolic heap to determine tree-ness of nodes:

**Lemma 3.3.** *Let  $SH$  be a symbolic heap and let  $CH$  be a corresponding concrete heap, defined by a function  $h$ . If a node  $u$  is the root of a tree in  $Graph(SH)$ , then  $h(u)$  also subtends a tree in  $CH$ .*

A proof gist is as follows. First, note that symbolic nodes and the node  $c_{nil}$  are always roots of trees in  $Graph(SH)$  and the locations in the concrete heap corresponding to them subtend trees (in fact, disjoint trees save the  $nil$  location). Turning to concrete nodes, we need to argue that if  $c$  is a concrete node in  $Graph(SH)$ , then  $h(c)$  is the root of a  $Dir$ -tree in  $CH$ . This follows by induction on the height of the tree under  $c$  in  $Graph(SH)$ , since each of the  $Dir$  children of  $c$  in  $Graph(SH)$  must either be the  $c_{nil}$  node or a summary node or a concrete node that is the root of a tree of less height. The corresponding locations in  $CH$ , by induction hypothesis or by the above observations, have  $Dir$ -trees suspended from them. In fact, by the definition of correspondence, these trees are all disjoint except for the  $nil$  location (since trees corresponding to summary nodes are all disjoint and disjoint from locations corresponding to concrete nodes, and since concrete nodes in the symbolic heap denote).

The location corresponding to a concrete node in  $Graph(SH)$  that does not have all  $Dir$ -fields defined in  $SH$  may or may not have a  $Dir$ -tree subtended from it; this is because the notion of correspondence allows the corresponding location to have more fields defined. In the sequel, when we use symbolic heaps for tracking footprints, such concrete nodes with partially defined  $Dir$  fields will occur only when processing function calls (where all information about a node may not be known).

### Initial footprint

Let the pre-condition be  $\varphi_{pre}(u, j_1, \dots, j_m)$ , where  $u$  is the only location program variable, there is a  $Dir$ -tree rooted at  $u$ , and  $j_1, \dots, j_m$  are integer program variables. Then we define the initial symbolic heap:

$$(C_0, S_0, I_0, c_{nil}, pf_0, df_0, pv_0)$$

where  $C_0 = \{c_{nil}\}$ ,  $S_0 = \{n_0\}$ ,  $I = \{i_1, \dots, i_m\}$ ,  $pf_0$  and  $df_0$  are empty functions (i.e. functions with an empty domain), and  $pv_0$  maps  $u$  to  $n_0$  and  $j_1, \dots, j_m$  to  $i_1, \dots, i_m$ , respectively. The initial formula  $\varphi_0$  is obtained from  $\varphi_{pre}(u, j_1, \dots, j_m)$  by replacing  $u$  by  $n_0$  and  $j_1, \dots, j_m$  by  $i_1, \dots, i_m$ , and by adding the conjunct  $p^*(c_{nil}) \leftrightarrow p_{base}$  or  $f^*(c_{nil}) = f_{base}$  for all recursive predicates and functions. Note that the formula is defined over the variables  $S_0 \cup I_0$ . Intuitively, we start at the beginning of the function with a single symbolic node that stands for the input parameter, which is a tree, and a concrete node that stands for  $nil$ . All integer parameters are assigned to distinct variables in  $I$ .

### Expanding the footprint

A basic operation on a pair,  $(SH; \varphi)$ , consisting of a symbolic heap, and a formula is expansion. Let  $SH$  be

$$(C, S, I, c_{nil}, pf, df, pv)$$

and  $n \in C \cup S$  be a node. We define  $expand((SH; \varphi), n) = (SH'; \varphi')$ , where  $SH'$  is the tuple

$$(C', S', I', c_{nil}, pf', df', pv')$$

as follows: if  $n \in C$  (the node is already expanded), then do nothing by setting  $(SH'; \varphi')$  to  $(SH; \varphi)$ ; otherwise:

- $C' = C \cup \{n\}$ , where  $n$  is the node being expanded

- $S' = S \uplus \{n_{dir} \mid dir \in Dir\} \setminus \{n\}$ , where each  $n_{dir}$  is a fresh new node different from the nodes in  $C \cup S$
- $I' = I \uplus \{i_f \mid f \in DF\}$ , where each  $i_f$  is a fresh new integer variable
- $pf' \upharpoonright_{C \setminus \{c_{nil}\} \times Dir} = pf$ , and  $pf'(n, dir) = n_{dir}$  for all  $dir \in Dir$
- $df' \upharpoonright_{C \setminus \{c_{nil}\} \times DF} = df$ , and  $df'(n, f) = i_f$  for all  $f \in DF$
- $pv' = pv$ ;

The formula  $\varphi'$  is obtained from the formula  $\varphi$  as follows:

$$\begin{aligned} \varphi' &= \varphi[\bar{p}_n, \bar{f}_n / \bar{p}^*(n), \bar{f}^*(n)] \\ &\wedge \bigwedge_{p^*} (p_n \leftrightarrow \hat{p}_{ind}(n)) \wedge \bigwedge_{f^*} (f_n = \hat{f}_{ind}(n)) \\ &\wedge n \neq c_{nil} \wedge \bigwedge_{\substack{n' \in C' \setminus \{c_{nil}\}, dir \in Dir}} (n' \neq n_{dir}) \\ &\wedge \bigwedge_{dir \in Dir, s \in S} (n_{dir} = s \rightarrow n_{dir} = c_{nil}) \\ &\wedge \bigwedge_{dir_1, dir_2 \in Dir, dir_1 \neq dir_2} (n_{dir_1} = n_{dir_2} \rightarrow n_{dir_1} = c_{nil}) \end{aligned}$$

where  $\bar{p}_n$  are fresh Boolean variables,  $\bar{f}_n$  are fresh term (integer, set, ...) variables,  $p^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil}, p_{base}, p_{ind}(x))$  ranges over all the recursive predicates, and  $f^*(x) \stackrel{def}{=} \text{ite}(x = \text{nil}, f_{base}, f_{ind}(x))$  ranges over all the recursive functions. Intuitively, The variables  $\bar{p}_n$  and  $\bar{f}_n$  capture the values of the predicates and functions for the node  $n$  in the current symbolic heap. This is possible because the values of the recursive predicates and functions for concrete non- $nil$  nodes are determined by the values of the functions and predicates for symbolic nodes and the  $nil$  node. The formula  $\hat{p}_{ind}(n)$  is obtained from  $p_{ind}(n)$ , by substituting every location term of the form  $n.dir$  with  $n_{dir}$  for every  $dir \in Dir$ , and substituting every integer term of the form  $n.f$  with  $i_f$  for every  $f \in DF$ . The term  $\hat{f}_{ind}(n)$  is obtained by the same substitutions.

### Evolving the footprint on basic blocks

Given a symbolic heap  $SH$  along with a formula  $\varphi$ , and a basic block  $bb$ . We compute the symbolic execution of  $bb$  using the transformation function  $st((SH; \varphi), bb) = (SH'; \varphi')$ . The transformation function  $st$  is computed transitively; i.e., if  $bb$  is of the form  $(stmt; bb')$  where  $stmt$  is an atomic statement and  $bb'$  is a basic block, then

$$st((SH; \varphi), bb) = st(st((SH; \varphi), stmt), bb')$$

Therefore, it is enough to define the transformation for the various atomic statements. Given  $SH = (C, S, I, c_{nil}, pf, df, pv)$ ,  $\varphi$  and an atomic statement  $stmt$ , we define  $st((SH; \varphi), stmt)$  as follows by cases of  $stmt$ . Unless some assumptions fail (in which case the transformation is undefined), we describe  $st((SH; \varphi), stmt)$  as  $(SH'; \varphi')$ .

As per our convention, function updates are denoted in the form of  $[arg \leftarrow new\_val]$ . For example,  $pv[u \leftarrow n]$  denotes the function  $pv$  except that  $pv(u)$  maps to  $n$ . Formula substitutions are denoted in the form of  $[new/old]$ . For example,  $\varphi[df'/df]$  denotes the formula obtained from the formula  $\varphi$  by substituting every occurrence of  $df$  with  $df'$ .

The following defines how the footprint evolves across all possible statements *except* function calls:

(a)  $stmt : u := v$

If  $pv(v)$  is undefined, the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I, c_{nil}, pf, df, pv[u \leftarrow pv(v)]) \\ \varphi' &\equiv \varphi \end{aligned}$$

(b)  $stmt : u := nil$

$$\begin{aligned} SH' &= (C, S, I, c_{nil}, pf, df, pv[u \leftarrow c_{nil}]) \\ \varphi' &\equiv \varphi \end{aligned}$$

(c)  $stmt : u := v.dir$

If  $pv(v)$  is undefined, or  $pv(v) \in C$  and  $pf(pv(v), dir)$  is undefined, the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand((SH; \varphi), pv(v))$$

Now  $pv''(v)$  must be in  $C'' \setminus \{c_{nil}\}$ , and we set

$$\begin{aligned} SH' &= (C'', S'', I'', c_{nil}, pf'', df'', pv''[u \leftarrow pf''(pv''(v), dir)]) \\ \varphi' &\equiv \varphi'' \end{aligned}$$

(d)  $stmt : j := v.f$

If  $pv(v)$  is undefined, or  $pv(v) \in C$  and  $pf(pv(v), f)$  is undefined, the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand((SH; \varphi), pv(v))$$

Now  $pv''(v)$  must be in  $C'' \setminus \{c_{nil}\}$ , and we set

$$\begin{aligned} SH' &= (C'', S'', I'' \uplus \{i\}, c_{nil}, pf'', df'', pv''[j \leftarrow i]) \\ \varphi' &\equiv \varphi'' \wedge i = df''(pv''(v), f) \end{aligned}$$

(e)  $stmt : u.dir := v$

If  $pv(u)$  or  $pv(v)$  is undefined, or  $pv(u) = c_{nil}$ , the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand((SH; \varphi), pv(u))$$

Now  $pv''(u)$  must be in  $C'' \setminus \{c_{nil}\}$ , and we set

$$\begin{aligned} SH' &= (C'', S'', I'', c_{nil}, pf''[(pv''(u), dir) \leftarrow pv''(v)], df'', pv'') \\ \varphi' &\equiv \varphi'' \end{aligned}$$

(f)  $stmt : u.f := j$

If  $pv(u)$  or  $pv(j)$  is undefined, or  $pv(u) = c_{nil}$ , the transformation is undefined. Otherwise we expand the symbolic heap:

$$((C'', S'', I'', c_{nil}, pf'', df'', pv''); \varphi'') = expand((SH; \varphi), pv(u))$$

Now  $pv''(u)$  must be in  $C'' \setminus \{c_{nil}\}$ , and we set

$$\begin{aligned} SH' &= (C'', S'', I'' \uplus \{i\}, c_{nil}, pf'', df''[(pv''(u), f) \leftarrow i], pv'') \\ \varphi' &\equiv \varphi'' \wedge i = pv''(j) \end{aligned}$$

(g)  $stmt : u := new$

We assume that, for the new location, every pointer initially points to `nil` and every data field initially evaluates to 0.

$$\begin{aligned} SH' &= (C \uplus \{n\}, S, I \uplus \{i_f \mid f \in DF\}, c_{nil}, pf', df', pv[u \leftarrow n]) \\ \varphi' &\equiv \varphi \wedge \bigwedge_{f \in DF} (i_f = 0) \wedge \bigwedge_{n' \in C \cup S} (n \neq n') \end{aligned}$$

where  $pf'$  and  $df'$  are defined as follows:

- $pf' \mid_{C \setminus \{c_{nil}\} \times Dir} = pf$ , and  $pf'(n, dir) = c_{nil}$  for all  $dir \in Dir$
- $df' \mid_{C \setminus \{c_{nil}\} \times DF} = df$ , and  $df'(n, f) = i_f$  for all  $f \in DF$

(h)  $stmt : j := aexpr(\bar{k})$

If  $pv$  is undefined on any variable in  $\bar{k}$ , then the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I \uplus \{i\}, c_{nil}, pf, df, pv[j \leftarrow i]) \\ \varphi' &\equiv \varphi \wedge i = aexpr[pv(\bar{k})/\bar{k}] \end{aligned}$$

(i)  $stmt : assume bexpr(\bar{v}, \bar{j})$

If  $pv$  is undefined on any variable in  $pv(\bar{v})$  or in  $pv(\bar{j})$ , then the transformation is undefined; otherwise

$$\begin{aligned} SH' &= SH \\ \varphi' &\equiv \varphi \wedge bexpr[pv(\bar{v}), pv(\bar{j})/\bar{v}, \bar{j}] \end{aligned}$$

(j)  $stmt : return u$

If  $pv(u)$  is undefined, the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I, c_{nil}, pf, df, pv[ret\_loc \leftarrow pv(u)]) \\ \varphi' &\equiv \varphi \end{aligned}$$

(k)  $stmt : return j$

If  $pv(j)$  is undefined, the transformation is undefined; otherwise

$$\begin{aligned} SH' &= (C, S, I \uplus \{i\}, c_{nil}, pf, df, pv[ret\_int \leftarrow i]) \\ \varphi' &\equiv \varphi \wedge i = pv(j) \end{aligned}$$

We can show that for any atomic statement that is not a function call, the above computes the strongest post of the footprint:

**Theorem 3.4.** *Let  $(SH; \varphi)$  be a footprint and let  $stmt$  be any statement that is not a function call. Let  $(SH'; \varphi')$  be the footprint obtained from  $(SH; \varphi)$  across the statement  $stmt$ , as defined above. Let  $C$  denote the set of all concrete heaps that correspond to  $SH$  and satisfy  $\varphi$ , and let  $C'$  be the set of all heaps that result from executing  $stmt$  from any concrete heap in  $C$ . Then  $C'$  is the precise set of concrete heaps that correspond to  $SH'$  and satisfy  $\varphi'$ .  $\square$*

### Handling function calls

Let us consider the statement  $u := f(v, \bar{j})$  on the pair  $(SH; \varphi)$ . Let  $f(w, \bar{k})$  be the function prototype and  $\varphi_{post}$  its post-condition. If  $pv(v)$  or any element of  $pv(\bar{j})$  is undefined, the transformation is undefined. We also assume that the checking of the pre-condition for  $f$  is successful; in particular,  $pv(v)$  and all the nodes reachable from it are roots of trees.

Recall that certain nodes of the symbolic heap can be determined to point to trees (as discussed earlier). For any node  $n \in C \cup S$ , let us define  $reach\_nodes(SH, n)$  to be the subset of  $C \cup S$  that is reachable from  $n$  in  $Graph(SH)$ . Let

$$\begin{aligned} N_C &= (reach\_nodes(SH, pv(v)) \cap C) \setminus \{c_{nil}\} \\ N_S &= reach\_nodes(SH, pv(v)) \cap S \end{aligned}$$

Intuitively,  $N_C$  and  $N_S$  are the concrete non-`nil` and the symbolic nodes affected by the call. Let  $n_{ret\_loc}$  be the node returned by  $f$ . Let  $N'$  be the set of nodes generated by the call:  $N' = \{n_{ret\_loc}, pv(v)\}$  if  $\varphi_{post}$  does not havoc  $old\_w$ , and  $N' = \{n_{ret\_loc}\}$  otherwise. The resulting symbolic heap is  $(C', S', I', c_{nil}, pf', df', pv')$ , where:

- $C' = C \setminus N_C$
- $S' = (S \setminus N_S) \cup N'$
- $I' = I$
- $pf' \mid_D = pf \mid_D$ , and  $pf'(n, dir)$  is undefined for all the pairs  $(n, dir) \in (C' \setminus \{nil\} \times Dir) \setminus D$ , where  $D \subseteq (C' \setminus \{nil\}) \times Dir$  is the set of pairs  $(n', dir')$  such that  $pf(n', dir') \in C' \cup S'$
- $df' = df \mid_{C' \setminus \{c_{nil}\} \times DF}$
- $pv' = pv[u \leftarrow n_{ret\_loc}]$

Intuitively, the concrete and symbolic nodes affected by the call are removed from the footprint (and get quantified in the DR<sub>YAD</sub> formula), with the possible exception of  $pv(v)$  (if  $\varphi_{post}$  does not havoc  $old\_w$ ,  $pv(v)$  becomes a symbolic node). The returned node is added to  $S$ . The  $pf$  and  $df$  functions are restricted to the new set of concrete nodes, and all the directions and program variables pointing to quantified nodes become undefined.

Let  $\psi_{post}$  be the post-annotating formula in  $\varphi_{post}$ , we define the following formulas

$$\begin{aligned} \varphi_1 &\equiv \overline{\varphi[pre\_call\_r_n / r^*(n)]} \\ &\wedge \bigwedge_{n \in N_C, r^*} (pre\_call\_r_n = \hat{r}_{ind}(n)) \\ \varphi_2 &\equiv \psi_{post}[\overline{pv(v)/old\_w}, \overline{pv(\bar{j})/old\_k}][n_{ret\_loc}/ret\_loc] \\ &\quad \overline{[pre\_call\_r_{pv(v)} / old\_r^*(pv(v))]} \end{aligned}$$

where  $n$  ranges over  $N_C \cup N_S$ ,  $r^*$  ranges over all the recursive predicates and functions;  $pre\_call_{\mathcal{L}_n}$  are fresh logical variables;  $\hat{r}_{ind}(n)$  is obtained from  $r_{ind}(n)$  by replacing  $n.dir$  with  $pf(n, dir)$  and  $n.f$  with  $df(n, f)$  for all  $dir \in Dir$ ,  $f \in DF$ , and then by replacing  $r^*(n')$  with  $pre\_call_{\mathcal{L}_{n'}}$  for all  $n' \in N_C \cup N_S$ ;  $r^*(n)$  is the vector of all the recursive predicates and functions on all  $n \in N_C \cup N_S$ . Intuitively, in  $\varphi_1$  we add logical variables that capture the values of the recursive predicates and functions for the nodes affected by the call. In  $\varphi_2$  we replace the program variables in the  $\psi_{post}$  with the actual nodes and integer variables, and we replace the old version of the predicates and functions on  $old\_w$  with the variables capturing those values. Then the resulting formula is

$$\varphi' \equiv \varphi_1 \wedge \varphi_2$$

The case of  $j := g(v, \bar{k})$  is similar.

**Example: Search in AVL trees** To illustrate the above procedure expands the symbolic heap and generates formulas, we present it working on the search routine of an AVL tree. Figure 2 shows the `find` routine, which searches in an AVL tree  $t$  and returns `true` if a key  $v$  is found. The pre-condition  $\varphi_{pre}$ , post-condition  $\varphi_{post}$ , and user-defined recursive sets and predicates are also shown in Figure 2. In Figure 3, we present graphically how the symbolic heap evolves for a particular execution path of the routine. At each point of the basic block, we also formally show the updated symbolic heap  $SH$  and the corresponding formula  $\varphi$ .

### Incorporating the post-condition

Finally, after capturing the program state after execution  $bb$  by a pair  $(SH; \varphi)$ , we incorporate the post-condition  $\varphi_{post}$ , which contains the annotating formula  $\psi$ , and generate a verification condition. We compute the set of tree-nodes in the footprint  $SH$  and compute the recursively defined predicates and functions on them. Let

$$\begin{aligned} SH &= (C, S, I, c_{nil}, pf, df, pv) \\ N &= (tree\_nodes(SH) \cap C) \setminus \{c_{nil}\} \\ \varphi_{vc} &\equiv \varphi \wedge \bigwedge_{n \in N, r^*} (vc_{\mathcal{L}_n} = \hat{r}_{ind}(n)) \end{aligned}$$

where  $vc_{\mathcal{L}_n}$  is fresh logical variables; and  $\hat{r}_{ind}(n)$  are obtained from  $r_{ind}(n)$  by replacing  $n.dir$  with  $pf(n, dir)$  and  $n.f$  with  $df(n, f)$  for all  $dir \in Dir$ ,  $f \in DF$ , and then by replacing  $r^*(n')$  with  $vc_{\mathcal{L}_{n'}}$  for all  $n' \in N$ . Intuitively,  $N$  are the non-`nil` concrete nodes that are tree roots, while  $\varphi_{vc}$  introduces variables that capture the values of the recursive predicates and functions on non-`nil` concrete nodes.

Let  $u$  and  $\bar{k}$  be the original program variables of  $bb$ . Let  $v$  and  $\bar{j}$  be the new program variables appearing in  $\psi$  (only when  $bb$  ends before a function call). Let  $N'$  be the set of nodes that should be the roots of disjoint trees, as required by  $\varphi_{post}$ :  $N' = \{pv(v)\}$  if  $\varphi_{post}$  mentions `tree(v)`;  $N' = \{n_0, pv(ret\_loc)\}$  if  $\varphi_{post}$  mentions `old_u##ret_loc`;  $N' = \{n_0\}$  if  $\varphi_{post}$  mentions `tree(old_u)`; if  $\varphi_{post}$  mentions `havoc(old_u)`,  $N' = \{pv(ret\_loc)\}$  or  $N' = \emptyset$  depending on the basic block is within a function returning a location or an integer. Let

$$\begin{aligned} \psi_{vc} &\equiv \psi[pv(v)/v][pv(\bar{j})/\bar{j}][pv(ret\_loc)/ret\_loc][pv(ret\_int)/ret\_int] \\ &\quad [\overline{vc_{\mathcal{L}_{pv(v)}}}/\overline{r^*(pv(v))}][\overline{vc_{\mathcal{L}_{pv(ret\_loc)}}}/\overline{r^*(pv(ret\_loc))}] \\ &\quad [n_0/old\_u][\bar{i}/old\_k][oldest_{\mathcal{L}_{n_0}}/old_{r^*(n_0)}][\overline{vc_{\mathcal{L}_{n_0}}}/\overline{r^*(n_0)}] \end{aligned}$$

where  $r^*$  ranges over all the recursive predicates and functions,  $n_0$  and  $\bar{i}$  are the initial node and integer variables, and  $oldest_{\mathcal{L}_{n_0}}$  is the variable capturing the value of  $r^*(n_0)$  in the pre-heap. The substitution of  $r^*(n)$  with  $vc_{\mathcal{L}_n}$  is only performed for nodes in  $N$ .

We should verify that:

- (1)  $N' \subseteq N \cup S \cup \{c_{nil}\}$ , that is, the nodes required by  $\varphi_{post}$  to be tree roots are indeed tree roots;

- (2)  $reach\_nodes(SH, n_1) \cap reach\_nodes(SH, n_2) \subseteq \{c_{nil}\}$ , for all  $n_1, n_2 \in N'$ , such that  $n_1 \neq n_2$ ;
- (3)  $(SH; \varphi_{vc}) \rightarrow \psi_{vc}$ , that is, the constraints on the current states imply the constraints required by  $\psi$ .

The first two are readily checkable. The last one asserts that any concrete heap that corresponds to the symbolic heap  $SH$  and satisfies  $\varphi_{vc}$  must also satisfy  $\psi_{vc}$ . Checking the validity of this claim is non-trivial (undecidable) and we examine procedures that can soundly establish this in the next section.

## 4. Proving the Verification Condition

In this section we consider the problem of checking the verification condition  $(SH; \varphi_{vc}) \rightarrow \psi_{vc}$  generated in Section 3. We first show a small syntactic fragment of DRYAD that is entirely decidable (without formula abstraction) by a reduction to the powerful decidable logic STRAND<sub>dec</sub>. We then turn to the abstraction schemes for unrestricted verification conditions, and show how to soundly reduce verification conditions to a validity problem of quantifier-free theories of sets/multisets of integers, which is decidable using state-of-the-art SMT solvers.

### 4.1 A Decidable Fragment of DRYAD

Given verification conditions of the form  $(SH; \varphi_{vc}) \rightarrow \psi_{vc}$ , where  $SH$  is a symbolic heap and  $\varphi_{vc}$  and  $\psi_{vc}$  are DRYAD formulas, the validity problem is in general undecidable. However, decision procedures for fragments are desirable, as when a program does not satisfy its specification, the decision procedure would *disprove* the program, and confirm it with a counterexample, which helps programmers debug the code. In this subsection, we identify a small decidable fragment, that is very restricted and practically useful for only a small class of specifications, called DRYAD<sub>dec</sub>. We omit the proof of decidability, in interest of space.

Let us fix a set of directions  $Dir$  and a set of data-fields  $DF$ . DRYAD<sub>dec</sub> does not only restrict the syntax of DRYAD, but also restricts the recursive integers/sets/multisets/predicates that can be defined. We first describe the ways allowed in DRYAD<sub>dec</sub> to define recursions as follows:

- Recursive integers are disallowed;
- For each data field  $f \in DF$ , a recursive set of integers  $fs^*$  can be defined as

$$fs^*(x) = \text{ite}(x = \text{nil}, \emptyset, \{x.f\} \cup \bigcup_{dir \in Dir} fs^*(x.dir))$$

- For each data field  $f \in DF$ , a recursive multiset of integers  $fms^*$  can be defined as

$$fms^*(x) = \text{ite}(x = \text{nil}, \emptyset_m, \{x.f\}_m \cup \bigcup_{dir \in Dir} fms^*(x.dir))$$

- Recursive predicates can be defined in the form of

$$p^*(x) = \text{ite}(x = \text{nil}, \text{true}, \varphi_p(x) \wedge \bigwedge_{dir \in Dir} p^*(x.dir))$$

where  $\varphi_p(x)$  is a *local formula* with  $x$  as the only free variable. Local formulas are DRYAD formulas disallowing set/multiset minus, `ite`, subset, equality between locations, positive appearance of belongs-to and and negative appearance of  $\leq$  between sets/multisets. Intuitively,  $p^*(x)$  is evaluated to true if and only if every node  $y$  in the subtree of  $x$  satisfies the local formula  $\varphi_p(y)$ , which can be determined by simply accessing the data fields of  $y$  and evaluating the recursive sets/multisets for the children of  $y$ .

The exclusion of recursive integers prevents us from expressing heights/cardinalities (which are required in many specifications).

There are however interesting algorithms on inductive data structures, like binary heaps, binary search trees and treaps, whose verification can be expressed in  $\text{DRYAD}_{dec}$ .

On the syntax of  $\text{DRYAD}$ ,  $\text{DRYAD}_{dec}$  does not allow to refer to any child or any data field, for any location, i.e., terms of the form  $lt.dir$  or  $lt.f$  are disallowed. Difference operations and subset relations between sets/multisets are also disallowed.

Overall,  $\text{DRYAD}_{dec}$  is the most powerful fragment of  $\text{DRYAD}$  that we could find that embeds into a known decidable logic, like  $\text{STRAND}_{dec}$ . However, it is not powerful enough for the heap verifica-

tion questions that we would like to solve. This motivates formula abstractions that we describe in the next section.

## 4.2 Proving $\text{DRYAD}$ using formula abstractions

In typical data-structure algorithms, a recursive algorithm manipulates the data-structure for a few bounded number of steps, and then recursively calls itself to process the rest of the inductive data-structure, and finally fixes the structure before returning.

As argued in Section 1, in recursive proofs of such algorithms, it is very often sufficient to assume that the recursively defined predicates, integers, sets of keys, etc. are *arbitrary*, or uninterpreted,

<pre> int find(node t, int v) {   if (t = NULL) return false;   tv := t.value;   if (v = tv) return true;   else if (v &lt; tv) { w := t.left;     r := find(w, v); }   else { w := t.right;     r := find(w, v); }   return r; } </pre>	$\begin{aligned} \varphi_{pre} &\equiv \text{avl}^*(t) \\ \varphi_{post} &\equiv \text{avl}^*(t) \wedge \text{keys}^*(t) = \text{keys}^*(\text{old\_t}) \wedge h^*(t) = h^*(\text{old\_t}) \wedge \\ &\quad \text{ret\_loc} \neq 0 \leftrightarrow v \in \text{keys}^*(t) \\ \text{avl}^*(x) &\stackrel{\text{def}}{=} \text{ite}(x = \text{nil}, \text{true}, \text{avl}_{ind}(x)) \\ \text{avl}_{ind}(x) &\stackrel{\text{def}}{=} \text{avl}^*(x.\text{left}) \wedge \text{avl}^*(x.\text{right}) \wedge \text{keys}^*(x.\text{left}) \leq \{v.\text{value}\} \wedge \{v.\text{value}\} \leq \text{keys}^*(x.\text{right}) \wedge \\ &\quad x.\text{height} = h^*(x) \wedge -1 \leq h^*(x.\text{left}) - h^*(x.\text{right}) \wedge h^*(x.\text{left}) - h^*(x.\text{right}) \leq 1 \\ \text{keys}^*(x) &\stackrel{\text{def}}{=} \text{ite}(x = \text{nil}, \emptyset, \text{keys}_{ind}(x)) \\ \text{keys}_{ind}(x) &\stackrel{\text{def}}{=} \text{keys}^*(x.\text{left}) \cup \{x.\text{value}\} \cup \text{keys}^*(x.\text{right}) \\ h^*(x) &\stackrel{\text{def}}{=} \text{ite}(x = \text{nil}, 0, h_{ind}(x)) \\ h_{ind}(x) &\stackrel{\text{def}}{=} 1 + \max(h^*(x.\text{left}), h^*(x.\text{right})) \end{aligned}$
--	---

Figure 2. AVL find routine (left); pre/post conditions and recursive definition of  $\text{avl}^*$ ,  $\text{keys}^*$ , and  $h^*$  (right).

Graphical representation of $SH$	Formal representation of $SH$	Formula $\varphi$
	$C = \{c_{nil}\}, S = \{n_0\}, I = \{i_1\}$ $dir = \emptyset, df = \emptyset$ $pv = \{t \mapsto n_0, v \mapsto i_1\}$	$\text{avl}^*(n_0)$
assume ( $t \neq \text{nil}$ )		
	$C = \{c_{nil}\}, S = \{n_0\}, I = \{i_1\}$ $dir = \emptyset, df = \emptyset$ $pv = \{t \mapsto n_0, v \mapsto i_1\}$	$\text{avl}^*(n_0) \wedge n_0 \neq c_{nil}$
tv := t.value;		
	$C = \{c_{nil}, n_0\}, S = \{n_1, n_2\}, I = \{i_1, i_2, i_3, i_4\}$ $dir = \{(n_0, \text{left}) \mapsto n_1, (n_0, \text{right}) \mapsto n_2\}$ $df = \{(n_0, \text{value}) \mapsto i_2, (n_0, \text{height}) \mapsto i_3\}$ $pv = \{t \mapsto n_0, v \mapsto i_1, tv \mapsto i_4\}$	$\text{avl}_{n_0} \wedge n_0 \neq c_{nil} \wedge$ $\text{avl}_{n_0} \leftrightarrow (\text{avl}^*(n_1) \wedge \text{avl}^*(n_2) \wedge i_3 = h_{n_0} \wedge \text{keys}^*(n_1) \leq \{i_2\} \wedge$ $\{i_2\} \leq \text{keys}^*(n_2) \wedge -1 \leq h^*(n_1) - h^*(n_2) \wedge h^*(n_1) - h^*(n_2) \leq 1) \wedge$ $\text{keys}_{n_0} = \text{keys}^*(n_1) \cup \{v\} \cup \text{keys}^*(n_2) \wedge$ $h_{n_0} = 1 + \max(h^*(n_1), h^*(n_2)) \wedge$ $n_0 \neq c_{nil} \wedge n_0 \neq n_1 \wedge n_0 \neq n_2 \wedge (n_1 = n_2 \rightarrow n_1 = c_{nil}) \wedge i_4 = i_2$
assume ( $tv \neq v$ ) assume ( $tv < v$ ) w := t.left;		
	$C = \{c_{nil}, n_0\}, S = \{n_1, n_2\}, I = \{i_1, i_2, i_3, i_4\}$ $dir = \{(n_0, \text{left}) \mapsto n_1, (n_0, \text{right}) \mapsto n_2\}$ $df = \{(n_0, \text{value}) \mapsto i_2, (n_0, \text{height}) \mapsto i_3\}$ $pv = \{t \mapsto n_0, v \mapsto i_1, tv \mapsto i_4, w \mapsto n_1\}$	$\text{avl}_{n_0} \wedge n_0 \neq c_{nil} \wedge$ $\text{avl}_{n_0} \leftrightarrow (\text{avl}^*(n_1) \wedge \text{avl}^*(n_2) \wedge i_3 = h_{n_0} \wedge \text{keys}^*(n_1) \leq \{i_2\} \wedge$ $\{i_2\} \leq \text{keys}^*(n_2) \wedge -1 \leq h^*(n_1) - h^*(n_2) \wedge h^*(n_1) - h^*(n_2) \leq 1) \wedge$ $\text{keys}_{n_0} = \text{keys}^*(n_1) \cup \{v\} \cup \text{keys}^*(n_2) \wedge$ $h_{n_0} = 1 + \max(h^*(n_1), h^*(n_2)) \wedge$ $n_0 \neq c_{nil} \wedge n_0 \neq n_1 \wedge n_0 \neq n_2 \wedge (n_1 = n_2 \rightarrow n_1 = c_{nil}) \wedge i_4 = i_2$ $\wedge i_4 \neq i_1 \wedge i_4 < i_1$
r := find(w, v); return r;		
	$C = \{c_{nil}, n_0\}, S = \{n_1, n_2\}$ $I = \{i_1, i_2, i_3, i_4, i_5, i_6\}$ $dir = \{(n_0, \text{left}) \mapsto n_1, (n_0, \text{right}) \mapsto n_2\}$ $df = \{(n_0, \text{value}) \mapsto i_2, (n_0, \text{height}) \mapsto i_3\}$ $pv = \{t \mapsto n_0, v \mapsto i_1, tv \mapsto i_4, w \mapsto n_1, r \mapsto i_5, \text{ret\_loc} \mapsto i_6\}$	$\text{avl}_{n_0} \wedge n_0 \neq c_{nil} \wedge$ $\text{avl}_{n_0} \leftrightarrow (\text{pre\_call\_avl}_{n_1} \wedge \text{avl}^*(n_2) \wedge i_3 = h_{n_0} \wedge$ $\text{pre\_call\_keys}_{n_1} \leq \{i_2\} \wedge \{i_2\} \leq \text{keys}^*(n_2) \wedge$ $-1 \leq \text{pre\_call\_h}_{n_1} - h^*(n_2) \wedge \text{pre\_call\_h}_{n_1} - h^*(n_2) \leq 1) \wedge$ $\text{keys}_{n_0} = \text{pre\_call\_keys}_{n_1} \cup \{v\} \cup \text{keys}^*(n_2) \wedge$ $h_{n_0} = 1 + \max(\text{pre\_call\_h}_{n_1}, h^*(n_2)) \wedge$ $n_0 \neq c_{nil} \wedge n_0 \neq n_1 \wedge n_0 \neq n_2 \wedge (n_1 = n_2 \rightarrow n_1 = c_{nil}) \wedge i_4 = i_2$ $i_4 \neq i_1 \wedge i_4 < i_1 \wedge \text{avl}^*(n_1) \wedge \text{keys}^*(n_1) = \text{pre\_call\_keys}_{n_1} \wedge$ $h^*(n_1) = \text{pre\_call\_h}_{n_1} \wedge i_5 \neq 0 \leftrightarrow i_1 \in \text{keys}^*(n_1) \wedge i_6 = i_5$

Figure 3. Expanding the symbolic heap and generating the formulas

when applied to the inductive hypothesis that the algorithm works correctly on the smaller tree that it calls itself on. This is very common in manual verification of these algorithms, and the footprint formula obtained in Section 3 rewords the recursive properties expressed directly in terms of recursive properties on the locations that the program makes recursive calls. Hence, in order to find a simple proof, we can replace recursive definitions on symbolic nodes as uninterpreted functions that map the symbolic trees to arbitrary integers, sets of integers, multisets of integers, etc., which we call formula abstraction (see [25–27] where such abstractions have been used).

To prove a verification condition  $(SH; \varphi) \rightarrow \psi$  using formula abstraction, we drop  $SH$ , and we replace recursive predicates on symbolic nodes by uninterpreted Boolean functions, replace recursive integer functions as uninterpreted functions that map nodes to integers, and replace recursive set/multiset functions with functions that map nodes to arbitrary sets and multisets. Notice that the constraints regarding the concrete and symbolic nodes in  $SH$  were already added to  $\varphi$ , during the construction of the verification condition. The formula resulting via abstraction is a formula  $\varphi_{abs} \rightarrow \psi_{abs}$  such that: (1) if  $\varphi_{abs} \rightarrow \psi_{abs}$  is valid, then so is  $(SH; \varphi) \rightarrow \psi$  (the converse may not necessarily be true); (2) checking  $\varphi_{abs} \rightarrow \psi_{abs}$  is decidable, and in fact can be reduced to QF\_UFLIA, the quantifier-free theory of uninterpreted functions and arithmetic.

The validity of the abstracted formula  $\varphi_{abs} \rightarrow \psi_{abs}$  over the theory of uninterpreted function, linear arithmetic, and sets and multisets of integers, is decidable. The fact that the quantifier free theory of ordered sets is decidable is well known. In fact, Kuncak et al. [13] showed that the quantifier-free theory of sets with cardinality constraints is NP-complete. Since we do not need cardinality constraints, we use a slightly simplified decision procedure that reduces formulas with sets/multisets using uninterpreted functions that capture the characteristic functions associated with these sets/multisets. We omit these details in the interest of space.

## 5. Experimental Evaluation

In this section, we demonstrate the effectiveness and practicality of the DRYAD logic and the verification procedures developed in this paper by verifying standard operations on several inductive data structures. Each routine was written using recursive functions and annotated with a pre-condition and a post-condition, specifying a set of partial correctness properties including both structural and data requirements. For each basic block of each routine, we manually generated the verification condition  $(SH; \varphi)$  following the procedure described in Section 3. Then we examined the validity of  $\varphi$  using the procedure described in Section 4.2. We employ Z3 [11], a state-of-the-art SMT solver, to check validity of the generated formula  $\varphi_D$  formula in the quantifier-free theory of integers and uninterpreted functions QF\_UFLIA. The experimental results are tabulated in Figure 4.

**The data-structures, routines, and verified properties:** Lists are trees with a singleton direction set. *Sorted lists* can be expressed in DRYAD. The routines `insert` and `delete` insert and delete a node with key  $k$  in a sorted list, respectively, in a recursive fashion. The routine `insertion-sort` takes a singly-linked list and sorts it by recursively sorting the tail of the list, and inserting the key of the head into the sorted list by calling `insert`. We check if all these routines return a sorted list with the multiset of keys as expected.

A *binary heap* is recursively defined as either an empty tree, or a binary tree such that the root is of the greatest key and both its left and right subtrees are binary heaps. The routine `max-heapify` is given a binary tree with both its left and right trees are binary heaps. If the binary-heap property is violated by the root, it swaps the root with its greater child, and then recursively max-heapifies

that subtree. We check if the routine returns a binary heap with same keys as that of the input tree.

The *treap* data-structure is a class of binary trees with two data fields for each node: *key* and *priority*. We assume that all priorities are distinct and all keys are distinct. Treaps can also be recursively defined in DRYAD. The `remove-root` routine deletes the root of the input treap, and joins the two subtrees descending from the left and right children of the deleted node into a single treap. If the left or right subtree of the node to be deleted is empty, the join operation is trivial; otherwise, the left or right child of the deleted node is selected as the new root, and the deletion proceeds recursively. The `delete` routine simply searches the node to be deleted recursively, and deletes it by calling `remove-root`. The `insert` routine recursively inserts the new node into an appropriate subtree to keep the binary-search-tree property, then performs rotations to restore the min-heap order property, if necessary. We check if all these routines return a treap with the set of keys and the set of priorities as expected.

An *AVL tree* is a binary search tree that is balanced: for each node, the absolute difference between the height of the left subtree and the height of the right subtree is at most 1. The main routines for AVL are `insert` and `delete`. The `insert` routine recursively inserts a key into an AVL tree (similar to a binary search tree), and as it returns from recursion it checks the balancedness and performs one or two rotations to restore balance. The `delete` routine recursively deletes a key from an AVL tree (again, similar to a binary search tree), and as it returns from the recursion ensures that the tree is indeed balanced. For both routines, we prove that they return an AVL tree, that the multiset of keys is as expected, and that the height increases by at most 1 (for `insert`), can decrease by at most 1 (for `delete`), or stays the same.

*Red-black trees* are binary search trees that are more loosely balanced than the AVL trees, and were described in Section 3. We consider the `insert` and `delete` routines. The `insert` routine recursively inserts a key into a red-black subtree, and colors the new node red, possibly violating the red-black tree property. As it returns from recursion, it performs several rotations to fix the property. If the root of the whole tree is red, it is recolored black, and all the properties are restored. The `delete` routine recursively deletes a key from a red-black tree, possibly violating the red-black tree property. As it returns from recursion, it again performs several rotations to fix it. For both routines, we prove that they return a red-black tree, that the multiset of keys is as expected, and the black height increases by at most 1 (for `insert`), decreases by at most 1 (for `delete`), or does not change.

The *B-tree* is a data structure that generalizes the binary search tree in that for each non-leaf node the number of children is one more than the number of keys stored in that node. The keys are stored in increasing order, and if the node is not a leaf, the key with index  $i$  is no less than any key stored in the child with index  $i$  and no more than all the keys stored in the child with index  $i + 1$ . For each node except the root, the number of keys is in some range (typically between  $T - 1$  and  $2T - 1$ ). A B-tree is balanced in that for each node, the heights of all the children are equal. To describe the B-tree in our logic, we need three mutually recursive predicates: one that describes a B-subtree, and two that describe a list of keys and children. The  $b\text{-subtree}^*(t)$  states that the number of keys stored in the node is in the required range, and that keys and children list satisfies either the  $key\text{-child-list}^*(l)$  predicate (if the node is not a leaf) or the  $key\text{-list}^*(l)$  predicate (if the node is a leaf). The  $key\text{-child-list}^*(l)$  states that either the list has only one element, and the child satisfies the  $b\text{-subtree}^*(t)$  predicate, or the list has at least two elements, the key in the head of the list is no less than all the keys stored in the child and no greater than the keys stored in the tail of the list, the child satisfies  $b\text{-subtree}^*(t)$ ,

Data Structure	#Ints	#Sets	#MSets	#Preds	Routine	#BB	Max. #Nodes in Footprint	Total Time (s)	Avg. Time (s) per VC	VC proved valid?
Sorted List	0	0	1	1	insert	4	3	0.24	0.06	Yes
					delete	3	3	0.17	0.06	Yes
					insertion-sort	3	4	0.11	0.04	Yes
Binary Heap	0	0	1	1	max-heapify	5	8	1.89	0.38	Yes
Treap	0	2	0	1	insert	7	6	4.06	0.58	Yes
					delete	6	4	0.81	0.14	Yes
					remove-root	7	8	2.96	0.42	Yes
AVL Tree	1	0	1	1	insert	11	8	1.45	0.13	Yes
					delete	18	7	2.13	0.19	Yes
Red-Black Tree	1	0	1	1	insert	19	8	1.93	0.11	Yes
					delete	24	7	3.22	0.14	Yes
B-Tree	2	0	1	2	insert	12	6	0.40	0.03	Yes
					find	8	3	0.12	0.02	Yes
Binomial Heap	1	0	1	3	delete-minimum	3	7	0.29	0.10	Yes
					find-minimum	4	6	1.81	0.45	Yes
					merge	13	7	17.38	1.37	Yes
Total						147				

**Figure 4.** Results of program verification (see details at <http://www.cs.illinois.edu/~madhu/dryad> )

and the height of the child is equal to the height of the tail (the height of a list is defined as the maximum height of a child). The predicate  $key\text{-list}^*(l)$  is similarly defined. We consider the `find` and `insert` routines. The `find` routine iterates over the list of keys, and recurses into the appropriate child, until it finds the key or it arrives to a leaf. The `insert` procedure is more complex, as it assumes that the node it is inserting into is non-full, and prior to recursion it might need to split the child. For both routines, we check that the tree after the call is a B-tree, that the multiset of keys has the expected value, and that the height of the tree stays the same, or increases by at most 1 (for `insert`).

As an advanced data structure, the *binomial heap* is described by a set of predicates defined mutually recursively: *binomial-tree\**, *binomial-heap\** and *full-list\**. Briefly, a binomial-heap of order  $k$  consists of a binary-tree of order  $k$  and a binary-heap of order less than  $k$ . A binomial-tree of order  $k$  is an ordered tree defined recursively: the root contains the minimum key, and its children compose a binomial-heap of order  $k - 1$ , satisfying the full-list property. A full-list of order  $k$  consists of a tree of order  $k$  and a full-list of order  $k - 1$ . The left-child, right-sibling scheme represents each binomial tree within a binomial heap. Each node contains its key; pointers to its leftmost child and to the sibling immediately to its right; and its degree. The roots of a binomial heap form a singly-linked list (also connected by the sibling pointer). We access the binomial heap by a pointer to the first node on the root list.

The `find-minimum` routine expects a nonempty binomial heap, and moves the tree containing the minimum key to the head of the list. It returns the original heap if it is a single tree. Otherwise, it calls `find-minimum` on its tail list, and appends the returned list to the head tree; then if keys of the roots of the first two trees are unordered, swaps the two trees. We check that `find-minimum` returns a binomial tree followed by a binomial heap, such that the root of the tree contains the minimum key, and the head of the binomial heap is either the first or the second root of the original heap. The `merge` routine merges two binomial heaps  $x$  and  $y$  into one. If one of the two heaps is empty, it simply returns the other one. Otherwise, if the heads of the two heaps are of the same order, it merges the two head trees into one, merges the two tail lists recursively, and returns the new tree followed by the new heap; if not, say,  $x.order > y.order$ , then it merges  $x.sibling$  and  $y$ , concatenates the head tree of  $x$  and the new heap in an appropriate way satisfying the binomial-heap property. We check that `merge` returns a binomial heap such that the keys are the union of the two input binomial heaps, and the order increases up to 1. The

`delete-minimum` routine is non-recursive. It simply moves the minimum tree  $m$  to the head by calling `find-minimum`, and obtains two binomial heaps: a list of the siblings of  $m$ , and a list of the children of  $m$ . Finally it merges the two heaps by `merge`. We check that `delete-minimum` returns a binomial heap with the multiset of keys as expected.

Figure 4 summarizes our experiments, showing the results of verifying 147 basic blocks across these algorithms. For each data structure, we report the number of integers, sets, multisets and predicates defined recursively. For each routine, we report the number of basic blocks, the number of nodes in the footprint, the time taken by Z3 to determine validity of all generated formulas, and the validity result proved by Z3.

Note that only the first three data-structures (sorted list, binary heap and treap) fit in the decidable fragment described in Section 4.1 as they do not require recursively defined integers. Furthermore, since the difference operations between sets/multisets are disallowed, we can check all functional properties for these data-structures except checking that the set of keys/priorities at the end of the each routine is as expected.

We are encouraged by the fact that all these verification conditions that were deterministically generated by the methodology set forth in this paper were proved by Z3 efficiently; this proved all these algorithms correct. To the best of our knowledge, this is the first terminating automatic mechanism that can prove such a wide variety of data-structure algorithms written using imperative programming correct (in particular, binomial heaps and the B-trees presented here have not been proven automatically correct).

The experimental results show that DRYAD is a very expressive logic that allows us to express natural and recursive properties of several complex inductive tree data structures. Moreover, our sound procedure tends to be able to prove many complex verification conditions.

## 6. Related Work

There is a rich literature on program logics for heaps. We discuss the work that is closest to ours. In particular, we omit the rich literature on general interactive theorem provers (like Coq [12]) as well as general software verification tools (like Boogie [2]) that are not particularly adapted for heap verification.

Separation logic [4, 19, 23] is one of the most popular logics for verification of heap structures. Many dialects of separation logic combine separation logic with inductively defined data-structures. While separation logic gives mechanisms to compositionally rea-

son with the footprint and the frame it resides in, proof assistants for separation logic are often heuristic and incomplete [4], though a couple of small decidable fragments are known [3, 17]. A work that comes very close to ours is a paper by Chin et al. [8], where the authors allow user-defined recursive predicates (similar to ours) and build a terminating procedure that reduces the verification condition to standard logical theories. While their procedure is more general than ours (they can handle structures beyond trees), the resulting formulas are quantified, and result in less efficient procedures. Bedrock [9] is a Coq library that aims at mostly automated (but not completely automated) procedures that requires some proof tactics to be given by the user to prove verification conditions.

In manual and semi-automatic approaches to verification of heap manipulating programs [4, 23, 24], the inductive definitions of algebraic data-types is extremely common, and proof tactics unroll these inductive definitions, do extensive unification to try to match terms, and find simple proofs. Our work in this paper is very much inspired by the kinds of manual heap reasoning that we have seen in the literature.

The work by Zee et al. [26, 27] is one of the first attempts at full functional verification of linked data structures, which includes the development of the ЯНОВ system that uses higher-order logics to specify correctness properties, and puts together several theorem provers ranging from first-order provers, SMT solvers, and interactive theorem provers to prove properties of algorithms manipulating data-structures. While many proofs required manual guidance, this work showed that proofs can often be derived using simple tactics like unrolling of inductive definitions, unification, abstraction, and employing decision procedures for decidable theories. This work was also an inspiration for our work, but we chose to concentrate on deriving proofs using *completely automatic and terminating procedures*, where unification, unrolling, abstraction, and decidable theories are systematically exploited.

One work that is very close to ours is that of Suter et al. [25] where decision procedures for algebraic data-types are presented with abstraction as the key to obtaining proofs. However, this work focuses on sound and complete decision procedures, and is limited in its ability to prove several complex data structures correct. Moreover, the work limits itself to functional program correctness; in our opinion, functional programs are very similar to algebraic inductive specifications, leading to much simpler proof procedures.

There is a rich and growing literature on completely automatic sound, complete, and terminating decision procedures for restricted heap logics. The logic LISBQ [14] offers such reasoning with restricted reachability predicates and quantification. While the logic has extremely efficient decision procedures, its expressivity in stating properties of inductive data-structures (even trees) is very limited. There are several other logics in this genre, being less expressive but decidable [1, 5, 6, 18, 20–22]. STRAND is a recent logic that can handle some data-structure properties (at least binary search trees) and admits decidable fragments [15, 16] by combining decidable theories of trees with the theory of arithmetic, but is again extremely restricted in expressiveness. None of these logics can express the verification conditions for full functional verification of the data-structures explored in this paper.

**Acknowledgements** This work is partially funded by NSF CAREER award #0747041 and NSA contract H98230-10-C-0294.

## References

- [1] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, volume 3385 of *LNCS*, pages 164–180. Springer, 2005.
- [2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO'05*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
- [3] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS'04*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.
- [4] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
- [5] N. Bjørner and J. Hendrix. Linear functional fixed-points. In *CAV'09*, volume 5643 of *LNCS*, pages 124–139. Springer, 2009.
- [6] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR'09*, volume 5710 of *LNCS*, pages 178–195. Springer, 2009.
- [7] A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.
- [8] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, in press, 2010.
- [9] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245. ACM, 2011.
- [10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.
- [11] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
- [12] INRIA. The coq proof assistant. URL <http://coq.inria.fr/>.
- [13] V. Kuncak, R. Piskac, and P. Suter. Ordered sets in the calculus of data structures. In *CSL'10*, volume 6247 of *LNCS*, pages 34–48. Springer, 2010.
- [14] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL'08*, pages 171–182. ACM, 2008.
- [15] P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In *SAS'11*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.
- [16] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *POPL'11*, pages 611–622. ACM, 2011.
- [17] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV'08*, volume 5123 of *LNCS*, pages 428–432. Springer, 2008.
- [18] G. Nelson. Verifying reachability invariants of linked structures. In *POPL'83*, pages 38–47. ACM, 1983.
- [19] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.
- [20] Z. Rakamarić, J. D. 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*, volume 4349 of *LNCS*, pages 106–121. Springer, 2007.
- [21] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an SMT framework. In *ATVA'07*, volume 4762 of *LNCS*, pages 237–252. Springer, 2007.
- [22] S. Ranise and C. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM'06*, pages 206–215. IEEE-CS, 2006.
- [23] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE-CS, 2002.
- [24] G. Rosu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST'10*, volume 6486 of *LNCS*, pages 142–162. Springer, 2010.
- [25] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL'10*, pages 199–210. ACM, 2010.
- [26] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI'08*, pages 349–361. ACM, 2008.
- [27] K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *PLDI'09*, pages 338–351. ACM, 2009.