Natural Proofs for Structure, Data, and Separation

Xiaokang Qiu  Pranav Garg  Andrei Stefanescu  P. Madhusudan
University of Illinois at Urbana-Champaign, USA
{qiu2,garg11,stefane1,madhu}@illinois.edu

Abstract
We propose natural proofs for reasoning with complex properties of the dynamically manipulated heap of a program that combines constraints on structure of the heap, data stored in the heap, and separation of regions within the heap. Natural proofs are a subclass of proofs that are amenable to completely automated reasoning and provide sound but incomplete procedures, and capture common reasoning tactics in program verification. We develop a dialect of separation logic over heaps, called Drvao, with recursive definitions that avoids explicit quantification, develop ways to reason with heaplets using logical set theory, and develop natural proofs for reasoning with Hoare-triples written in Drvao using unfoldings and formula abstractions. We also implement the technique and show that a large class of correct programs over lists, cyclic lists, doubly-linked lists, and trees are amenable to automated proofs using the proposed natural proof method.

1. Introduction
In recent years, the software verification paradigm that combines user written modular annotations for specifications as well as invariants (pre/post conditions, loop invariants, assertions, ghost code, etc.) and automatic theorem proving of the resulting verification conditions has become very powerful. The latter process is often executed by logical decision procedures such as SMT solvers, which have emerged as robust and powerful engines to automatically find proofs. Several techniques and tools have been developed [2, 16, 19] and there have been several success stories of large software verification projects using this approach.

Verification conditions for straight-line code, however, do not always fall into decidable theories. In particular, the verification of properties of the dynamically modified heap is a very big challenge for logical methods. The dynamically manipulated heap poses several challenges, as typical correctness properties of heaps require complex combinations of structure (e.g., p points to a tree structure, or a doubly-linked list, or an almost balanced tree, with respect to certain pointer-fields), data (the integers stored in data-fields of the tree respect the binary search tree property, or the data stored in a tree is a max-heap), and separation (the procedure modifies only elements reachable using certain pointer-fields, etc.). The fact that the dynamic heap contains an unbounded number of locations means that expressing the above properties requires quantification, which immediately precludes the use of most SMT decidable theories (there are only a few of them known with quantification; e.g., the array property fragment [12] and the Strand logic [23, 24]).

There are roughly two ways to tackle the proving of verification conditions. The first is to build a proof system for a logic over heaps, and manually guide the proof using what essentially amounts to proof tactics. This is a prevalent technique in verification using classical logics; for example, in tools like VCC [16] that use Booan [2], reachability, tree-ness, etc. need to be modeled using new ghost variables that break down the property to locally definable properties, which need to get updated properly, and furthermore universal quantification, when used, is broken down by user-given instantiation tactics. User-suggested proof tactics are also prevalent in verification using separation logic; for example, in tools like Bedrock [15] and Verifast [19], where the user specifies low-level tactics at the code-level to guide the proof.

The second approach is to identify a subclass of verification conditions for which the validity problem is decidable. This approach leads to completely automatic methods, but the restrictions on the verification conditions are often severe. Several decidable logics are known: the array property fragment [12], Lsaq [21], CSL[9], Strand[23, 24], a decidable separation logic on lists [3], etc., but their expressive power is very limited. None of these logics can reason with the correctness of procedures manipulating a red-black tree or an AVL tree, for example, which involves balanced trees and the binary search tree constraints on the data.

The natural proof methodology:
The natural proof method, which we first proposed last year at POPL [25], is a third option, that exploits a fixed set of proof tactics, keeps the expressiveness of powerful logics, keeps the automated nature of proving validity, but gives up on completeness (i.e., gives up decidability, retaining soundness). The idea is to identify a subclass of proofs \( \mathcal{N} \) such that (a) a large class of valid verification conditions of real-world programs have a proof in \( \mathcal{N} \), and (b) searching for a proof in \( \mathcal{N} \) is decidable. In fact, we would like the search for a proof in \( \mathcal{N} \) to be efficiently decidable, possibly utilizing the automatic logic solvers (SMT solvers) that exist today. Natural proofs are hence a fixed set of proof tactics whose application is itself expressible in a decidable logic.

In previous work, we identified a class of natural proofs for tree data-structures. We exhibited a logic Drvao (called simply Drvao in [25]) which is a logic with recursive definitions over trees, and showed proof tactics that unfolded recursive definitions precisely across the footprint that a straight-line program manipulated, and then used formula abstraction, which interprets recursive definitions on the frontier of the footprint using uninterpreted functions. This work could handle only tree data-structures, as the check for tree-ness of nodes in the heap is determined using a graph-theoretic procedure independent of the reduction to decidable logics.

The aim of this paper is to provide natural proofs for general properties of structure, data, and separation. In particular, we propose Drvao, a dialect of separation logic for heaps, with no explicit quantification, but with recursive definitions to express second-order properties, and provide an effective class of natural proofs for them. We now motivate our decision for using separation logic as a means to find natural proofs for general structures beyond trees.

Logics over heaplets:
The primary design principle behind separation logic is the decision to express strict specifications—logical formulas must natu-
rally refer to heaplets (subparts of the heap), and, by default, the smallest heaplets over which the formula needs to refer to. This is in contrast to classical logics (such as FOL) which implicitly refer to the entire heap globally. Strict specifications permit elegant ways to capture how a particular sub-part of the heap changes due to a program, implicitly leaving the rest of the heap and its properties unchanged across a call to a procedure. Separation logic is a particular framework for strict specifications, where formulas are implicitly defined on strictly defined heaplets, and where heaplets can be combined using a spatial conjunction operator denoted by $\&$. The frame rule in separation logic captures the main advantage of strict specifications: if the Hoare-triple $[P] C \{Q\}$ holds for some program $C$, then $[P \& R] C \{Q \& R\}$ also holds (with side-conditions that the modified variables in $C$ are disjoint from the free variables in $R$).

While the above motivation for separation logic based on strict specifications is worthy in itself, separation logic syntax gives another distinct advantage to our goals of building natural proofs for generalized structural properties. Going from handling tree structures in our earlier work [25] to more general structures expressed in logic, a primary concern is how the structural property of the heap is expressed. Consider, for example, expressing that the location $x$ is the root of a tree. This is a second-order property and formulations of it in classical logic using set or path quantification make it much less amenable to verification using natural proofs. We prefer inductive definitions of structural properties without any explicit quantification. The separation logic syntax with recursive definitions and heaplet semantics allows simple quantifier-free formulas to express structural restrictions; for example, tree-ness can be expressed simply as:

$$\text{tree}(x) ::= (x = \text{nil} \land \text{emp}) \lor (x \rightarrow (l, r) \land \text{tree}(l) \land \text{tree}(r))$$

The above formulation lends well to the natural proof methodology — it doesn’t use quantification (the implicit quantification on $l$ and $r$ are special since they are uniquely determined by $x$) and it is amenable to unfolding across a footprint, and hence amenable to natural proofs, provided we can only handle the separation logic semantics in a decidable theory.

**Dxvad and natural proofs for heaplets and separation:**

One of the primary contributions of this paper is a systematic method to reason with heaplets and their separation using the theory of sets.

We first define a new logic, Dxvad, that permits no explicit quantification, but permits powerful recursive definitions in order to define integers, sets/multisets of integers, and sets of locations, using least fixed-points. The logic Dxvad furthermore has a heaplet semantics and allows the spatial conjunction operator $\&$. However, a key design feature of Dxvad is that the heaplet for recursive formulas is essentially determined by the syntax as opposed to the semantics. For instance, general formulations of separation logic would allow a formula’s heaplet to be a particular path in a tree determined by the data stored in the tree. This kind of power is often not necessary, and furthermore hinders automatic reasoning, as it introduces an implicit quantification of heaplets. In a verification condition that combines the pre-condition in the positive and the post-condition in the negative, such a semantics invariably introduces universal quantification in the satisfiability query of the negation of the verification condition, which in turn is extremely hard to handle. In Dxvad, formula semantics are required to have essentially syntactically determined heaplets (e.g., a typical property of a tree would require the entire tree to be part of the heaplet).

We then show that heaplet semantics and separation logic constructs can be effectively translated to classical logic where heaplets are modeled as sets of locations. We show that Dxvad formulas can be translated into classical logic with free set variables that capture the heaplets corresponding to the strict semantics. This translation does not, of course, yield a decidable theory yet, as recursive definitions are still present.

We then show a natural proof methodology for Dxvad over straight-line programs. This idea is similar in spirit to our earlier work on natural proofs [25]: to unfold recursive definitions precisely across footprints, translating them to the frontier of the footprint, and then use formula abstraction that treats recursive formulas on frontier nodes as uninterpreted functions in order to realize it in a decidable logic. The key new feature is that heaplets and separation logic constructs, which get translated to recursively defined sets of locations, are unfolded and formula abstracted using the same natural proof strategy, leading to proving generic properties that go beyond trees.

While our proof strategy is roughly as above, there are many more technical details to fill in. For example, the heaplets defined by pre/post conditions intrinsically specify the modified locations of the heap, which have to be considered when processing procedure calls in order to ensure which recursively defined metrics on locations continue to hold after a procedure call. The final decidable theories that we compile our conditions down to does require a bit of quantification, but it turns out to be in the array property fragment which has reasonably efficient decision procedures.

**Implementation and Evaluation:**

Our proof mechanisms are essentially a class of decidable proof tactics that result in sound but incomplete validation procedures. In order to show that this class of natural proofs is effective in practice, we provide a prototype implementation of our technique, which compiles a straight-line code manipulating a heap with pre-conditions and post-conditions written in Dxvad to a formula that is in a decidable theory solvable using a standard SMT solver. We show using a large class of correct programs manipulating lists, trees, cyclic lists, and doubly linked lists that the natural proof mechanism succeeds in proving the verifications conditions automatically. Furthermore, in several cases, when the annotations supplied were incorrect, the model provided by the SMT solver proved useful in detecting the error and correcting the invariants.

In summary, this paper builds natural proof mechanisms (sound, terminating, but incomplete proof tactics that are formalizable in decidable logics) for a powerful logic with heaplet semantics that can express properties of structure, data, and separation. We define a new quantifier-free dialect of separation logic with recursive definitions, called Dxvad, develop natural proof mechanisms for it, and show its efficacy in proving several programs correct using a prototype implementation. In terms of our earlier work on natural proofs [25], the results herein generalize to logically expressed structural properties written in separation logic, and are not limited to trees. From the perspective of the literature on separation logic, our methods give a fairly generic way of exploiting decidable logics and SMT solvers in order to prove powerful properties automatically, using carefully chosen restrictions that allow capturing heaplet semantics using decidable theories of sets.

2. Related Work

Apart from our work in [25] on natural proofs for tree data-structure, there is a rich literature on program logics for heaps. We discuss the work closest to ours. In particular, we omit the rich literature on general interactive theorem provers (like Isabelle [28]) that require considerable manual guidance. We also omit a lot of work on analyzing shape properties of the heap [6, 13, 18, 26, 37], as they do not handle complex functional properties.
There are several proof systems and assistants for separation logic [29, 33] that incorporate proof heuristics and are incomplete. However, [3] gives a small decidable fragment of separation logic on lists which has been further extended in [11] to include a restricted form of arithmetic. Symbolic execution with separation logic has been used in [4, 5, 8] to prove structural specifications for various list and tree programs. These tools come hardwired with a collection of axioms and their symbolic execution engines check the entailment between two formulas modulo these axioms. VERIFAST [19] on the other hand, chooses the flexibility of writing richer specifications over complete automation, but requires the user to provide some inductive lemmas and proof tactics to aid verification. Similarly, BÖROCK [15] 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. A work that comes very close to ours is a paper by Chin et al. [14], 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. However, their procedure does not search for a proof in a well-defined simple and decidable class, unlike our natural proof methodology; in fact, the resulting formulas are quantified, are not compatible with decidable logics handled by SMT solvers.

In all of the above cited work and other manual and semi-automatic approaches to verification of heap-manipulating programs like [34], inductive definitions of algebraic data-types is extremely common for capturing second-order data-structure properties. Most of these approaches use proof tactics which unroll inductive definitions and do extensive unification to try to match terms to find simple proofs. Our notion of natural proofs is very much inspired by the such kinds of semi-automatic and manual heap reasoning that we have seen in the literature.

There is also a variety of verification tools based on classical logics and SMT solvers. DANNY [22] and VCC [16] compile to BooCox [2] and generates VCs that are passed to SMT solvers. This approach requires significant ghost annotations, and annotations that explicitly express and manipulates frames. The Jaorn system [38, 39] is one of the first attempts at full functional verification of linked data structures, which integrates a variety of theorem provers, including SMT solvers, and makes the process mostly automated. However, complex specifications combining structure, data and separation usually require more complex provers such as MONA [20], or even interactive theorem provers such as Isabelle [28] in the worst case. The success of the search algorithm also relies on users’ manual guidance.

The idea of unfolding recursive definitions and formula abstraction also features in the work by Suter et al. [35, 36], where a procedure for algebraic data-types is presented. However, this work focuses on soundness and completeness, and is not terminating for several complex data structures like red-black trees. 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 also a rich literature on completely automatic decision procedures for restricted heap logics, some of which combine structure-logic and arbitrary data-logic. These logics are usually FOLs with restricted quantifiers, and usually are decided using SMT solvers. The logics Linsos [21] and CSL [9, 10] offer such reasoning with restricted reachability predicates and quantification; see also the logics in [1, 7, 27, 30–32]. STRAND is a relatively expressive logic that can handle some data-structure properties (at least BSTs) and admits decidable fragments [23, 24], but is again not expressive enough for more complex properties of inductive data-structures. None of these logics can express the variety of logics that can support for full functional verification explored in this paper.

```c
void heapify(loc x) {
    if (x.left == nil) 
        s := x.right;
    else if (x.right == nil)
        s := x.left;
    else {
        lx := x.left;
        rx := x.right;
        if (lx.key < rx.key)
            s := x.right;
        else
            s := x.left;
    }
    if (s.key > x.key) {
        t := x.key;
        x.key := s.key;
        heapify(s);
    }
}
```

Figure 1. Motivating example: Heapify

3. Motivating Example

In this section we give intuition into our verification approach through a motivating example. Recall that a max-heap, or simply a heap, is a binary tree such that for each node, the key stored at a node is greater than or equal to the keys stored at each of its children. Max-heaps are often used to implement priority queues. In Figure 1, we express the property that a location `x` points to a max-heap using recursive definitions `keys*(x)` and `heap*(x)`. These recursive definitions are written in Dvyao, which is formally introduced in Section 4. Intuitively, Dvyao extends quantifier free separation logic [29, 33] with recursive predicates and functions. These recursive definitions allow us to express structural and data properties on the heap, like `heap*(x)` and `keys*(x)`, without explicit quantification.

The recursive definition `keys*(x)`, for any location `x`, returns the set of keys in the nodes of the tree rooted at `x`. If `x` is nil, it is the empty set; otherwise `keys*(x)` is the union of the key stored at `x` and the keys stored under the left and the right children of `x`. The recursive definition `heap*(x)` states that location `x` points to a max-heap if `x` is nil and the heaplet is empty; or the heaplet at the left child of `x` and the heaplet at the right child of `x` are mutually disjoint, which are both disjoint from the location `x`, i.e., `x` points to a tree. Further, it requires that the key at `x` is greater than or equal to the keys of the left and the right children of `x`.

The method `heapify` in Figure 1 is at the heart of the procedure for deleting the root from the max-heap (removing the node with the maximum priority). In the figure, we present the pre and post conditions of the method, `φpre` and `φpost` respectively. If the heap property is violated at a node `x` while the property is satisfied by its children, then the method `heapify` restores the heap property at `x`. It does so by recursively descending into the tree, swapping the key of the root with the key at its left or right child, whichever is greater. The precondition `φpre` binds the free variable `K` to the keys at `x`. The postcondition states that after the procedure call, `x` satisfies the heap property and the keys at `x` is unchanged; it is same as `K`.

One of the main aspects of our approach is to reduce reasoning about heaplet semantics and separation logic constructs to reasoning about sets of locations. We use set operations like union, intersection and membership to describe separation constraints on a heaplet satisfying a formula. This translation from Dvyao formu-
las, like those in Figure 1, to quantifier-free monadic second order logic formulas is formally presented in Section 5. Intuitively, to each (spatial) atomic formula we associate a set of locations, which is the domain of the heaplet satisfying that formula. Dryad requires that this heaplet is syntactically determined for each formula. For example, the heaplet associated to the formula $x \rightarrow \ldots$ is the singleton $\{x\}$; for recursive definitions like $heap^p(x)$ and $keys^s(x)$, the domain of the heaplet is $reach(left,right)(x)$ which is intuitively the set of locations reachable from $x$ using the pointer fields, namely the left and right pointers in our example.

From Figure 1, the pre-condition $\phi_{pre}$ is a conjunction of two sub-formulas. If $G_{pre}$ is the domain of the heaplet satisfying $\phi_{pre}$, then the first conjunct requires $G_{pre}$ to be the disjoint union of $\{x\}$, $reach(left(x))$ and $reach(right(x))$, i.e., $G_{pre} = \{x\} \cup reach(left(x)) \cup reach(right(x))$. From the second conjunct, $G_{pre} = reach(x)$. From these heaplet constraints, we can translate the pre-condition $\phi_{pre}$ to the following formula in classical logic over the global heap:

$$x \neq nil \land G_{pre} = \{x\} \cup reach(left(x)) \cup reach(right(x))$$

$$\land heap(left(x)) \land heap(right(x)) \land keys(x) = K$$

Similarly, we translate $\phi_{post}$ to

$$G_{post} = reach(x) \land heap(x) \land keys(x) = K$$

Note that we use the recursive definition $heap$ and $keys$ without * are interpreted in classical semantics (without the heaplet constraint). Hence the recursive predicate $heap$ satisfies

$$heap(x) \leftrightarrow (x = nil \land reach(x) = \emptyset)$$

$$\lor (heap(left(x)) \land \{keys(left(x))\land heap(right(x)) \land \{keys(right(x))\land reach(x) = \{x\} \cup reach(left(x)) \cup reach(right(x))\})$$

Now consider the right side of Figure 1, there is a basic path from the method heapify by, with the subscript of a pointer/data field denoting the timestamp, corresponding to the case when both children of $x$ are not nil and the key of the left child is greater than the key of the right child and the root. A key insight is that any basic path touches a finite number of locations and may make some calls to recursive procedures. We call the touched locations the footprint, and the adjacent locations which are not part of the footprint the frontier. For this example, the footprint is $\{x, left_0(x), right_0(x)\}$ and the frontier is $\{left_0(left_0(x)), right_0(left_0(x)), left_0(right_0(x)), right_0(right_0(x))\}$

We capture the effect of the path until the call to heapify by

$$left_0(x) \neq nil \land right_0(x) \neq nil \land lx = left_0(x) \land rx = right_0(x) \land key_0(lx) < key_0(rx) \land k = right_0(x) \land k \neq nil$$

$$\land key_0(y) > key_0(x) \land t = key_0(x)$$

$$\land key_0 = key_0(x) \leftarrow key_0(y) \land key_2 = key(y \leftarrow t)$$

A key issue is how to track the recursive predicates and functions across a basic path. We need to evaluate these recursive definitions at the beginning and the end of the path, and also before and after every procedure call to incorporate the effect of the procedure call. We “compute” these by expressing the definitions of them on nodes within the footprint using their recursive definitions. Furthermore, at frontier locations, we know that if the corresponding reach set from that location hasn’t changed due to the basic path working on the footprint (i.e., the reach sets from the location do not involve footprint nodes), then the recursive definitions on the frontier do not change. Similarly, if a procedure is called and its pre-condition defines a heaplet that is disjoint from the reach-set of a location, then we can retain the value for a recursive definition for that location across a call to the procedure; otherwise it will have to be updated conservatively taking into account the pre/post condition of the called procedure. All these conditions involving heaplets, reach-sets, scoping, modified-sets, etc., can be formulated in logic using set theory, and Section 5 describes this in detail. Finally, the verification condition is written using a logic over the global heap, but referring only to the footprint, and the recursive definitions are formula-abstracted, resulting in a formula in a decidable theory, whose proof is then attempted.

4. The Logic Dryad

We present a logic that redefines the logic Dryad as in [25] on arbitrary data-structures (not just trees), using heaplet semantics and separation logic primitives; the logic hence is a quantifier-free heaplet logic augmented with recursively defined predicates/functions and ordered sets. We will continue to refer to this logic as Dryad, and call the logic in [25] as Dryad$_{pre}$.

4.1 Syntax

Let us fix a finite set of pointer-fields $PF$ and a finite set of data-fields $DF$. A record consists of a set of pointer-fields from $PF$ and a set of data-fields from $DF$. Our logic also presumes that locations refer to entire records rather than particular fields, so that address arithmetic is precluded. We will use the term locations hence to refer to these records.

Let $Bool = \{true, false\}$ 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 $S(A)$ denote the powerset of $A$, and let $MS(A)$ denote the set of all finite multisets with elements in $A$.

The Dryad logic allows expressing quantifier-free first-order properties over heaps/heaplets augmented with five kinds of recursively defined notions for a location to express second-order properties:

- recursively defined integer functions ($Loc \rightarrow Int$),
- recursively defined set-of-locations functions ($Loc \rightarrow S(Loc)$),
- recursively defined set-of-integers functions ($Loc \rightarrow S(Int)$),
- recursively defined multiset-of-integers functions ($Loc \rightarrow MS(Int)$), 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 set of locations, recursively defined set/s/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 of nodes reachable from a location following certain pointer fields (recursively defined sets of locations), the set/multiset of keys stored at a particular data-field under nodes reachable from a location (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 or just a tree (recursively defined predicates).

A Dryad formula consists of a pair $\langle Def, \phi \rangle$, where $Def$ is a set of recursive definitions and $\phi$ is a formula. The syntax of Dryad logic is given in Figure 2, 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 $\phi$ has a unique definition in $Def$. The definitions are parameterized by a set of program variables $\vec{r}$. The recursively defined functions are defined using the syntax:

$$f^p(x_1, \ldots, x_n, \vec{r}) : t_1^p(x_1, \vec{r}), \ldots, t_n^p(x_1, \vec{r}), \text{default:} t_n^p(x_1, \vec{r})$$

where each $f^p$ is a set of pointer fields, $\vec{r}$ is a set of location terms, and $f^p(x_1, \ldots, \vec{r})/t_n^p(x, \vec{r})$ is a formula/term in our logic with $\vec{r}$ implicitly existentially quantified. The recursively defined predicates are
defined using the syntax: \( \varphi^t(x, \bar{r}, \bar{v}) \), which is a formula in our logic with \( \bar{v} \) implicitly existentially quantified. The restrictions on these recursive definitions are:

- Subtraction, set-difference, and negation are disallowed;
- Every variable in \( \bar{v} \) should appear in the right hand side of a points-to relation binding it to \( x \) exactly once.

4.2 Semantics

We extend the primitive data-types to lattice domains, in order to give least fixed-point semantics for recursive definitions in the logic. Boolean with the order \( \text{false} \subseteq \text{true} \) forms a complete lattice, and \( S(\text{Loc}) \) and \( S(\text{Int}) \) ordered by inclusion, with join as union and meet as intersection, form complete lattices. Integers and multisets are extended to lattices. Let \( (\text{Int}_\text{c}, \preceq) \) denote the complete lattice, where \( \text{Int}_\text{c} = \text{Int} \cup \langle -\infty, \infty \rangle \), and where the ordering is \( \preceq \), join is \( \max \), meet is \( \min \). Also, \( S(\text{Int}_\text{c}, \subseteq) \) denote the complete lattice constructed from \( S(\text{Int}) \), where \( S(\text{Int}_\text{c}) = S(\text{Int}) \cup \langle \top \rangle \), and \( \subseteq \) extends the inclusion relation with \( S \subseteq T \) for any \( M \in S(\text{Int}) \). It is easy to see that \( S(\text{Int}_\text{c}, \subseteq) \) and \( S(\text{Int}(\text{c}, \langle \top \rangle) \) are complete lattices.

Our logic is interpreted on models that are program states:

**Definition 4.1.** A program state is a tuple \( C = (R, s, h) \) where

- \( R \subseteq \text{Loc} \setminus \{\text{nil}\} \) is a finite set of locations;
- \( s : \text{Vars} \to \text{Int} \cup \text{Loc} \) is a store mapping program variables to locations or integers (appropriate type);
- \( h : R \times (PF \cup DF) \to \text{Int} \cup \text{Loc} \) is a heap mapping non-nil locations and each pointer-field/data-field to values of the appropriate type.

Note that the set of locations is, in general, larger than the state \( R \) and hence \( R \) defines a subset of heap locations. The store maps variables to locations (not necessarily in \( R \), but the heap function \( h \) gives interpretations for pointer and data-fields only for elements in \( R \).

Given a heap function \( h \), for every pointer field \( pf \), we denote the projection of \( h \) on \( R \times (PF \setminus \{pf\} \cup DF) \) as \( h_f pf \); similarly, for every data-field \( df \), we denote the projection of \( h \) on \( R \times (PF \cup DF \setminus \{f\}) \) as \( h_f df \). Also, for every subset \( S \subseteq R \), we denote the projection of \( h \) on \( S \times (PF \cup DF) \) as \( h_f S \).

The role of \( \bar{pf} \) and \( \bar{r} \) in a recursive definition \( f^\bar{r}\bar{pf} \) is to delinate the heaplet over which the recursive definition is defined. Intuitively, the heaplet for \( f^\bar{r}\bar{pf} \) is the set of all locations reachable from \( f \) using the pointer-fields in \( \bar{pf} \), but that do not pass through the locations denoted by the terms in \( \bar{r} \). We formalize this using reach-sets, as defined below.

For a program state \( (R, s, h) \), a location \( l \), a set of pointer-fields \( \bar{pf} \), and a set of terms \( \bar{r} \), we would like to define a set \( \text{reach}_\bar{r} \bar{pf} \) that is precisely the set of locations reachable from \( f \) using pointer-fields in \( \bar{pf} \), but \emph{without} going through the locations denoted by \( \bar{r} \). In other words, we want to take the set of locations that lie \emph{in between} \( l \) and \( l' \).

Formally, we define \( \text{reach}_\bar{r} \bar{pf} (l, (R, s, h)) \) as the smallest set of locations \( L \) satisfying the following two conditions:

1. \( l \) is in the set if \( l \) is not in \( \bar{r} \) and \( l \neq \text{nil} \);
2. for each \( c \in L \) and for each pointer \( pf \), if \( h(c, pf) \) is not in \( \bar{r} \) and is not \( \text{nil} \), then \( h(c, pf) \) is also in \( L \).

For each recursive definition \( \text{rec} \), we denote the heaplet corresponding to the definition as \( \text{reach}^\text{rec} \bar{r} \bar{pf} \bar{df} (l, (R, s, h)) \), where this set is defined to be \( \text{reach}_\bar{r} \bar{pf} \bar{df} (l, (R, s, h)) \), where the definition of \( \text{rec} \) is of the form \( \text{rec}^\text{rec} \bar{r} \bar{pf} \bar{df} f(x) = \ldots \). Notice that the heaplet of a recursively defined formula \( \text{rec}^\text{rec} \bar{r} \bar{pf} \bar{df} f(x) \) is fixed to be the set of all nodes reachable on paths starting from \( x \) till the terms \( \bar{r} \), and does not change depending on the semantics of the recursive definition for \( \text{rec}^\text{rec} \).

A formula with free variables \( F \) is interpreted by interpreting the free variables in \( F \) according to an interpretation.

Before defining the semantics of formulas, we define the heapless property for any term/formula. Intuitively, a term/formula is heapless if it can be evaluated only using the store. Given a term \( t \) we define heapless \((t) \) inductively in Figure 3. Note that formulas with recursive definitions are not heapless, nor is any formula with separating conjunction \( \ast \). We are now ready to define the semantics under a model \( C = (R, s, h) \).

### Semantics for terms (non-recursive)

Each \( T \)-term evaluates to either a normal value of type \( T \), or to undefined. The \( Loc \) terms are evaluated as follows:

\[
\begin{align*}
\textit{true} & \in C = s(x) \\
\textit{false} & \in C = \text{nil}
\end{align*}
\]

For any other operator \( \text{op} \), \( \text{op} t \) is evaluated as follows:
Figure 3. The heapless predicate for term formulas. The $f^*$ function refers to any recursive integer/set/multi-set definition. The op and $\sim$ operators stand for appropriate operators.

$$\llbracket t \text{ op } f^* \rrbracket_c = \begin{cases} \llbracket t \rrbracket_c \text{ op } \llbracket f^* \rrbracket_c, & \text{if heapless}(C, t) \text{ or heapless}(C, f^*) \\ \llbracket t \rrbracket_c \llbracket f^* \rrbracket_{c_{RJ}}, & \text{else if there exist } R_1, R_2 \text{ such that } R = R_1 \cup R_2, \llbracket t \rrbracket_{c_{RJ}} \neq \text{undef} \\ \text{undef}, & \text{otherwise} \end{cases}$$

where op is interpreted in the natural way.

For singletons, $[it]$ will evaluate to 0 if it evaluates to $-\infty$ or $\infty$:

$$\llbracket [it] \rrbracket_c = \begin{cases} \text{undef}, & \text{if } [it]_c = \text{undef} \\ 0, & \text{if } [it]_c = -\infty \text{ or } \infty \\ [it]_c \end{cases}$$

$[it]_m$ will evaluate similarly.

Semantics for formulas (non-recursive)

The semantics of formulas is defined as follows. The formula true is always interpreted to be true:

$$(R, s, h) \models \text{true}$$

The formula emp asserts that the heap is empty:

$$(R, s, h) \models \text{emp} \iff R = \emptyset$$

The formula $lt_{pf} \llbracket f^* \rrbracket_{c_{RJ}}$ asserts that the heap contains exactly one record consisting of fields $pf$ and $df$, at address $lt$, with values $lt$ and $it$, respectively:

$$(R, s, h) \models \llbracket t \llbracket_{\text{RJ}, t} \llbracket f^* \rrbracket_{c_{RJ}} \iff R = \llbracket t \rrbracket_{\text{RJ}, t} \text{ and } h(\llbracket t \rrbracket_{\text{RJ}, t}, pf) = \llbracket t \rrbracket_{\text{RJ}, t, pf} \text{ for corresponding } pt \text{ and } lt, \text{ and } h(\llbracket t \rrbracket_{\text{RJ}, t, df}) = \llbracket t \rrbracket_{\text{RJ}, t, df} \text{ for corresponding } dt \text{ and } it.$$}

Note that, as in separation logic, the above has a strict semantics—the heaplet must be a singleton set and cannot be a larger set.

All other relations between integers, sets, and multisets, including the equality, are interpreted as follows:

$$(R, s, h) \models t \sim t' \iff \text{heapless}(t) \land \text{heapless}(t') \land \llbracket t \rrbracket_c \sim \llbracket t' \rrbracket_c \lor \sim \text{heapless}(C, t) \land \sim \text{heapless}(C, t')$$

there exist $R_1, R_2$ such that $R = R_1 \cup R_2$ and $\llbracket t \rrbracket_{c_{RJ}} \neq \text{undef}, \llbracket t \rrbracket_{c_{RJ}} \neq \text{undef} \land \llbracket t \rrbracket_{c_{RJ}} \sim \llbracket t' \rrbracket_{c_{RJ}}$, where $\sim$ is interpreted in the natural way.

The semantics of the disjoint conjunction operator $*$ is defined as follows. The formula $\varphi_0 * \varphi_1$ asserts that the heap can be split into two disjoint parts in which $\varphi_0$ and $\varphi_1$ hold respectively:

$$(R, s, h) \models \varphi_0 * \varphi_1 \iff \text{there exist } R_0, R_1 \text{ such that } R_0 \cap R_1 = \emptyset \text{ and } R_0 \cup R_1 = R \text{ and } (R_0, s, h \mid R_0) \models \varphi_0 \text{ and } (R_1, s, h \mid R_1) \models \varphi_1$$

Semantics of recursive definitions

The semantics of recursively defined functions/predicates are defined in the natural way. Let Def consists of definitions of integer functions $I$, set-of-locations functions $SL$, set-of-integers functions $SI$, multiset-of-integer functions $MSI$ and predicates $P$. Since these definitions could rely on each other, we evaluate them altogether as a function vector

$$r^* = (I^*, SL^*, SI^*, MSI^*, P^*)$$

We take the cartesian product lattice of the individual lattices and take the least-fixed point of $r^*$ to obtain the semantics for each definition. Let select($lt_{pf}(r^*)$), for each recursive definition $f^*$, denote the selection of the coordinate for $f^*$ in $lt_{pf}(r^*)$.

For any $C = (R, s, h)$, the semantics of a recursive function $f^*$ is defined as:

$$f^*(lt)_{R,s,h} = \begin{cases} \text{select}(lt_{pf}(r^*(R, s, h))), & \text{if } R = \text{reach}^l((lt)_c, h) \\ \text{undef}, & \text{otherwise} \end{cases}$$

and the semantics of a recursive predicate $p^*$ is defined as:

$$p^*(lt)_{R,s,h} = \begin{cases} \text{select}(lt_{pf}(r^*(R, s, h))), & \text{if } R = \text{reach}^h((lt)_c, h) \\ \text{false}, & \text{otherwise} \end{cases}$$

Note that the heaplet for a recursive definition $f^*_z (x)$ is reach$^h (x)$, which is precisely reach$^h (x)$, the set of all nodes reachable on paths starting from $x$, taking pointer fields in $f$, and ending with nil or terms in $f^*$. Examples

The Dvano logic was already used in Section 2 to define max-heaps. Note that the definition of a max-heap is defined precisely defined on the heaplet that includes the underlying tree nodes of the max-heap only, as the heaplet for a recursive definition is the set of all reachable nodes according to the two pointers.

Consider now this recursive definition on locations of a binary search tree:

$$p^*_z(x, y, z) \equiv (x = \text{nil} \land \text{emp} \land ((y \llbracket f \rrbracket \rightarrow d \cdot \text{true}) \land k < x \cdot y \land (x \llbracket f \rrbracket \rightarrow (y, z) \cdot p^*(y)))) \lor (x \llbracket f \rrbracket \rightarrow (y, z) \cdot p^*(z))$$

The right-hand side of this definition, when expressed in classical separation logic, would be defined precisely on the set of nodes constituting the path chosen while looking for the key $k$ in the tree, and is determined by the data stored in the tree. However, in Dvano, the semantics of $p^*$ is that the heaplet be the entire tree starting from the node $x$. When we check a verification condition along the lines of $p^*(x) \land \ldots \Rightarrow p^*(t)$ (a precondition implying a postcondition), the corresponding satisfiability query would be $p^*(x) \land \ldots \land -p^*(t)$. The negated formula $\neg p^*(t)$ hence requires a universal quantification over paths in the tree if we went with the usual separation logic semantics, which is very hard to handle automatically. Consequently, in our semantics, the heaplet for $p^*$ is defined to be a fixed heaplet, based only on its signature. Another stark example is: $q_{z_1}(x) \equiv (x = \text{nil} \land \text{emp} \land ((x \llbracket f \rrbracket \rightarrow (y, z) \cdot (q^*(y) \cdot \text{true}) \lor (q^*(z) \cdot \text{true})) \lor \neg p^*(t))$ where classical separation logic would evaluate true over an unbounded number of heaplets corresponding to paths in a tree, while the heaplet in Dvano would be the entire tree.

Dvano can express structures beyond trees. The main restriction we do impose is that we allow only unary recursive definitions, as this allows us to find simpler natural proofs since there is only one
way to unfold the definition across a footprint. However, Drayd can express structures like cyclic lists and doubly-linked lists.

A cyclic-list is captured as $h \mapsto y * lseg(y)$. Here, $h$ is a program variable which denotes the head of the cyclic-list and $lseg(y)$ captures the list segment from $y$ back to the head $h$.

$$lseg(y) \overset{def}{=} (y = h \land \text{emp}) \lor (y \mapsto z * lseg(z))$$

Another interesting example is a doubly-linked list. We define a doubly-linked list as the following unary predicate:

$$dll^*(x) = \begin{cases} x = \text{nil} \land \text{emp} \lor \quad \text{next} \\ x \mapsto y \land y = \text{nil} \lor \quad \text{prev} \\ \{ (x \mapsto y \land y \mapsto x \land \text{true}) \land \{ x \mapsto y \land dll^*(y) \} \} \end{cases}$$

The first two disjuncts in the definition cover the base case when $x$ is nil or the location $y$, next to $x$ is nil; if $y$ is not nil then the prev pointer at $y$ points to $x$ and location $y$ is recursively defined as a doubly-linked list.

5. Translation to a logic over the global heap

We now show one of the main contributions of this paper—a translation from a large fragment of separation logic extended with recursive predicates and functions to classical logic with recursive predicates and functions, but over the global heap. The formulation of separation logic primitives in the global heap allows us to express complex structural properties, like disjointness of heaplets and tree-ness, to recursive definitions over sets of locations, which are defined locally, and are amenable to unfolding across the footprint and hence amenable to natural proofs.

For example, consider the formula $\text{tree}^*(x) * \text{tree}^*(y)$, since the heaplets for $\text{tree}^*(x)$ and $\text{tree}^*(y)$ are precise, it can get translated to the equivalent formula with a free set variable $G$ that denotes the global heap over which the formula is evaluated:

$$\text{tree}(x) \land \text{tree}(y) \land (\text{reach}_{\{x\}}(x) \land \text{reach}_{\{y\}}(y) = \emptyset) \land (\text{reach}_{\{x\}}(x) \lor \text{reach}_{\{y\}}(y) = G)$$

where $\text{tree}$ and $\text{reach}_{\{x\}}$ are corresponding recursive definitions in classical logic, which will be defined later in this section.

We assume the Drayd formula to be translated in the disjunctive normal form, i.e., $\lor \land \lor$ should be above all $\lor$ and $\land$ operators. This is just a syntactic restriction as one can always push the disjunction out. This normal form ensures that for all occurrences of the separation operator in a formula, there exists a unique way of splitting the heap so as to satisfy the $*$ separated sub-formulas. Also, it ensures that this unique heap-split can be determined syntactically from the structure of those sub-formulas.

In our translation, we model the heaplets associated with a formula or a term as a set of locations and all operations on these heaplets are modeled as set operations like set union, set intersections, etc., over set-of-location variables. For example the separa-


\[ P ::= P ; P \mid \text{stmt} \]

\[ \text{stmt} ::= u := v \mid u := \text{nil} \mid u := v.pf \mid u.pf := v \mid j := u.df \mid u.df := j \mid \text{def} \mid u := \text{new} \mid \text{free } u \mid \text{assume bexpr} \]

\[ \text{aexpr} ::= \text{int} \mid j \mid \text{aexpr + aexpr} \mid \text{aexpr - aexpr} \]

\[ \text{bexpr} ::= u = v \mid u = \text{nil} \mid \text{aexpr} \leq \text{aexpr} \mid \text{aexpr} \geq \text{aexpr} \mid \text{bexpr} \lor \text{bexpr} \]

\[ \text{reach}^{\text{rec}}(x) \triangleq \text{ITE}(x = \text{nil}, 0, \text{ITE}(x \in R, \{ u \mid \bigcup_{p \in \text{Pr}(\text{rec})} (\text{reach}^{\text{rec}}(p(x)), \{ |x| \}) \}) \]

We denote the righthand side as reachDF^{\text{rec}}(x). For each recursive predicate \( p^* \) defined as \( p^*(x) \triangleq \phi^*(x,\tilde{z},\tilde{v}) \), we define \( p(x) \triangleq T^*(\phi^*(x,\tilde{z},\tilde{v}), \text{reach}^*(x)) \).

We define the righthand side as def*(x,\tilde{z},\tilde{v}). Similarly, for each recursive function \( f^* \) defined as

\[ f^*(x) \triangleq \text{ITE}(\phi_1^*(x,\tilde{z},\tilde{v}), t_1(x,\tilde{v}) \mid \ldots \mid \phi_k^*(x,\tilde{z},\tilde{v}), t_k(x,\tilde{v}) \mid \text{otherwise : } t_{k+1}(x,\tilde{v})) \]

Then we define

\[ f(x) \triangleq \text{ITE}(\phi_1^*(x,\tilde{z},\tilde{v}), \text{reach}^*(u), t_1(x,\tilde{v}) \mid \text{ITE}(\phi_2^*(x,\tilde{z},\tilde{v}), \text{reach}^*(u), t_2(x,\tilde{v}) \mid \ldots \mid \text{ITE}(\phi_{k+1}^*(x,\tilde{z},\tilde{v}), \text{reach}^*(u), t_{k+1}(x,\tilde{v}) \ldots)) \]

We denote the righthand side as def(x,\tilde{z},\tilde{v}). Now for each set of recursive definitions Def in Drvad, we can translate it to a set of recursive definitions Def* in classical logic.

**Theorem 5.1.** Let Def be a set of recursive definitions and let \( \varphi \) be a Drvad formula. For every program state C and every interpretation of variables I including a set-variable \( G, (C, I) \models \phi(G) \) w.r.t. Def* if and only if \( (C \cup I \setminus G) \models \varphi \) w.r.t. Def. \( \square \)

### 6. Natural Proofs for Drvad

In this section we show how Drvad can be used in reasoning about the correctness of imperative heap-manipulating programs, in terms of verifying Hoare-triples where the pre- and post-conditions are expressed in Drvad. We first introduce a simple programming language and the corresponding Hoare-triples. Then we present the natural proof framework which consists of three steps. In the first step, we generate a classical logic formula as the verification condition, utilizing in part the translation defined in Section 5. In the second step, we utilize the idea of unfolding across the footprint strengthen the verification condition. Finally, we prove the validity of the VC soundly using the technique of formula abstraction.

#### 6.1 Programs and Hoare-triples

We consider straight-line program segments that do destructive pointer-updates, data-updates and recursive procedure calls. Parameterized by a set of pointer fields PF and a set of data-fields DF, the syntax of the programs is defined in Figure 7, where \( pf \in PF, f \in DF, u \) and \( v \) are program variables of type location, \( j \) and \( z \) are program variables of type integer, \( int \) is an integer constant. To simplify the presentation, we assume all program variables are local and are either pre-assigned or assigned once in the program.

We allow two kinds of recursive procedures, one returning a location \( f(\tilde{v}, \tilde{z}) \) and one returning an integer \( g(\tilde{v}, \tilde{z}) \). Each procedure/program is annotated with its pre- and post-conditions in Drvad. The pre-condition is denoted as a formula \( \psi_{\text{pre}}(\tilde{v}, \tilde{z}, \tilde{c}) \), where \( \tilde{v} \) and \( \tilde{z} \) are variables as the formal parameters/program variables, \( \tilde{c} \) is a set of complimentary variables, which are implicitly existentially quantified. The post-condition is denoted as a formula \( \psi_{\text{post}}(\text{ret loc}, \tilde{v}, \tilde{z}, \tilde{c}) \), where \( \text{ret} \) is the variable representing the return value, of corresponding type, \( \tilde{v} \) and \( \tilde{z} \) are program variables, \( \tilde{c} \) is a set of complimentary variables that have appeared in the pre-condition \( \psi_{\text{pre}} \).

Given a straight-line program with its pre- and post-conditions

\[ \{ \psi_{\text{pre}} \} \quad \text{P} \quad \{ \psi_{\text{post}} \} \]

we define its partial correctness without considering memory errors:\footnote{We exclude memory errors in order to simplify the presentation. Memory errors can be handled using a similar VC generation for assertions that negate the conditions for memory errors to occur.} P is partially correct iff for every normal execution (memory-error free) of P, which transits state C to state \( C' \), if \( C \models \psi_{\text{pre}} \), then \( C' \models \psi_{\text{post}} \).

#### 6.2 Generating the Verification Condition

Given a Hoare-triple \( \{ \psi_{\text{pre}} \} \quad \text{P} \quad \{ \psi_{\text{post}} \} \) as defined above, in the presence of a set of recursive definitions and a set of annotated procedure declarations, we show how to algorithmically derive the verification condition corresponding to it. Assume that P consists of n statements, then consider a normal execution \( E \), which can be represented as a sequence of program states \( (S_0, \ldots, S_n) \), where \( S_i = (R_i, s_i, h_i) \) represents the program state after executing the first \( i \) statements. The verification condition is just a formula interpreted on a state sequence \( (S_0, \ldots, S_n) \). Let \( pf_j : \text{Loc} \to \text{Loc} \) be the function mapping every location \( l \) to its pf pointer, i.e., \( pf_j(l) = h_j(l, pf) \) for every location \( l \). Similarly, \( df_j : \text{Loc} \to \text{Int} \) is defined such that \( df_j(l) = h_j(l, df) \) for every l. Recall that every program variable is either pre-assigned or assigned once in the program, each \( s_j \) is an expanding of the previous one, and \( s_j \) is the store with all program variables defined. Hence we simply use \( v \) to denote \( s_i(v) \).

Moreover, every recursive predicate/function is also indexed by \( i \). For example, \( p_i \) is the recursive predicate such that \( p_i(l) \) is true iff \( C_i \models T(p_i(l), \text{reach}^i(l)) \). Now for every formula \( \varphi \) and every index \( i \), we can give the index \( i \) to all the pointer fields, data fields and recursive definitions. We denote the indexed formula as \( \varphi[i] \).

Assume there are \( m \) procedure calls in \( P \), then \( P \) can be divided into \( m + 1 \) basic segments (subprograms without procedure calls):

\[ S_0 ; g_1 ; S_1 ; \ldots ; g_m ; S_m \]

where \( S_d \) is the \( d + 1 \)-th basic segment and \( g_d \) is the \( d \)-th procedure call.

For each \( d \in [m] \), let the \( d \)-th procedure call in \( P \) be the \( t_d \)-th statement (we also extend the index \( d \) to 0 and \( m + 1 \) such that \( t_0 = 0 \) and \( t_{m+1} = n + 1 \)). Note that \( E \) requires that a portion of the state \( C_{t_d-1} \) satisfies the precondition of the call, and a portion of the state \( C_{t_d} \) satisfies the postcondition of the call. We denote the two required portions \( C_{t_d-1} \cap \text{Call}_{d} \) and \( C_{t_d} \cap \text{Return}_{d} \), respectively, where \( \text{Call}_{d} \subseteq R_{t_d-1} \) and \( \text{Return}_{d} \subseteq R_{t_d} \) are two sets of records.

Let all the location variables appearing in \( P \) be \( L \text{Vars} \). We call a location variable \( v \) dereferenced if \( v \) appears on the left-hand side of a dereferencing operator \( * \) in \( P \). We call a location variable \( v \) modified if \( v \) appears in a statement of the form \( v.pf := u \) or \( v.df := j \) in \( P \). Then we can extract the set of dereferenced variables \( \text{Deref} \) and the set of modified variables \( \text{Mod} \). Note that a modified variable is always dereferenced, i.e., \( \text{Mod} \subseteq \text{Deref} \). For each basic segment \( S_d \), let the dereferenced and modified variables within the segment be \( \text{Deref}_{d} \) and \( \text{Mod}_{d} \), respectively.

For the \( d \)-th procedure call, let the pre- and post-condition associated with the procedure be \( \psi_{\text{pre}}^{d}(\tilde{v}, \tilde{z}, \tilde{c}) \) and \( \psi_{\text{post}}^{d}(\text{ret loc}, \tilde{v}, \tilde{z}, \tilde{c}) \),...
\begin{align*}
[u := v] & \quad \psi_1 \equiv u = v \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
[u := \text{nil}] & \quad \psi_1 \equiv u = \text{nil} \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
[u := \text{v.pf}] & \quad \psi_1 \equiv v \in R_{i-1} \land u = \text{pf}_i(v) \land R_i = R_{i-1} \\
  & \quad \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
[u.pf := v] & \quad \psi_1 \equiv u \in R_{i-1} \land \text{pf}_i = \text{pf}_i(v \leftarrow u) \land R_i = R_{i-1} \\
  & \quad \land \text{FieldsUnmod}(PF \cup \{\text{pf}_i\} \setminus \text{df}, i, i-1) \\
[j := u.df] & \quad \psi_1 \equiv u \in R_{i-1} \land j = \text{df}_{i-1}(u) \land R_i = R_{i-1} \\
  & \quad \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
[j := \text{acexpr}] & \quad \psi_1 \equiv j = \text{acexpr} \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
[u := \text{new}] & \quad \psi_1 \equiv \text{new}_i \neq \text{nil} \land u = \text{new}_i \land \text{new}_i \notin R_{i-1} \land R_i = R_{i-1} \cup \{\text{new}_i\} \\
  & \quad \land \bigwedge_{\text{df}} (\text{pf}_i = \text{pf}_i(\text{new}_i)) \land \bigwedge_{\text{df}} (\text{df}_j = \text{df}_{j-1}(0 \leftarrow \text{new}_i)) \\
[\text{free u}] & \quad \psi_1 \equiv u \in R_{i-1} \land R_i = R_{i-1} \setminus \{u\} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\text{assume bexpr} & \quad \psi_1 \equiv \text{bexpr} \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
[u := f(\vec{v}, \vec{z})] & \quad \psi_1 \equiv T(\text{pf}_i(\vec{v}, \vec{z}, c_d), \text{Call}_i)[i-1] \land T(\text{pf}_i(u, \vec{v}, \vec{z}, c_d), \text{Return}_i)[i] \\
  & \quad \land (R_{i-1} \setminus \text{Call}_i) \cap \text{Return}_i = \emptyset \land R_i = (R_{i-1} \setminus \text{Call}_i) \cup \text{Return}_i \\
  & \quad \text{where } d \text{ is the index such that } \text{df}_i = i \\
[j := g(\vec{v}, \vec{z})] & \quad \psi_1 \text{ is defined in the same way as the above case,} \\
  & \quad \text{except replacing } u \text{ with } j.
\end{align*}

where \(\text{FieldsUnmod}(F, i, j)\) is short for \(\bigwedge_{\text{df}} (\text{field}_i = \text{field}_j)\).

**Figure 8.** Formulas capture modification by statements respectively. Since \(E\) is a normal execution, we have \(C_{i-1} \models T(\text{pf}_i(\vec{v}, \vec{z}, c_d), \text{Call}_i)\) and \(C_i \models T(\text{pf}_i(u, \vec{v}, \vec{z}, c_d), \text{Return}_i)\) (assume the procedure call returns a location to \(u\)), where \(\vec{v}\) and \(\vec{z}\) are the actual parameters of the procedure call, \(c_d\) are the complimentary variables with fresh names.

Now we are ready to define the verification condition corresponding to \(P\). We first derive a formula expressing that \(E\) does not involve null pointer dereference:

\[\text{NonNullDereference} \equiv \bigwedge_{v \in \text{Def}} v \neq \text{nil} \]

For each \(i \in [n]\), Figure 8 shows the effect of each statement on the verification condition generated. Each statement’s strongest post condition is captured in the logic, and for procedure calls, the heaplet manipulated by the procedure is carefully taken into account to update the heap at the caller. The conjunction of these formulas captures the modification made in \(E\):

\[\text{Modification} \equiv \bigwedge_{i \in [n]} \psi_i\]

Finally, we can define two formulas to capture the pre- and post-conditions:

\[\text{Pre} \equiv T(\text{pf}_0, R_0)[0]\]
\[\text{Post} \equiv T(\text{pf}_n, R_n)[n]\]

Now the validity of \(P\) can be captured by the following formula:

\[\psi_{\text{VC}} \equiv (\text{Pre} \land \text{NonNullDereference} \land \text{Modification}) \rightarrow \text{Post}\]

**Theorem 6.1.** Given a Hoare-triple \([\text{pf}_0 \mid P \mid \text{pf}_n]\), assume that each procedure call in \(P\) satisfies its associated pre- and post-conditions. Then the triple is valid if the formula \(\psi_{\text{VC}}\) derived above is valid. Moreover, when \(P\) contains no procedure calls, the triple is valid if \(\psi_{\text{VC}}\) is valid. \(\square\)

### 6.3 Unfolding Across the Footprint

The verification condition obtained above is a quantifier-free formula involving recursive definitions and the reachable sets of the form \(\text{reach}^{\text{re}}(x)\), which is also defined recursively. Intuitively, the footprint is the set of locations explored by the program explicitly (not including procedure calls). More precisely, a location is in the footprint if it is dereferenced explicitly in the program. While these recursive definitions can be unfolded ad infinitum, the idea of unfolding across the footprint is to unfold the recursive definitions precisely over the footprint of the program, so that recursive definitions on the frontier nodes are related, as tightly as possible, to the recursive definitions on frontier nodes. This will enable effective use of the formula abstraction mechanism, as when recursive definitions on frontier nodes are made uninterpreted, the formula ensures tight conditions that the frontier nodes have to satisfy.

Let \(u\) be a location variable in \(\text{LVars}\) and let \(i\) be an timestamp such that \(1 \leq i \leq n\). For each recursive definition \(\text{rec}_i\) whose \(\ast\)-eliminated version defined as \(\text{pf}_i \equiv \text{def}^{\text{rec}_i}(x, i, \vec{v})\) and whose reach set defined as \(\text{reach}^{\text{rec}_i}(x) \equiv \text{def}^{\text{rec}_i}(x, i, \vec{v})\), we can derive a formula \(\text{Unfold}^{\text{rec}_i}(i, u)\) for unfolding both \(\text{rec}_i\) and its corresponding reach set on \(u\) at timestamp \(i\), provided that \(u\) is allocated at the current timestamp \((u \in R_i)\). Let \(\text{def}^{\text{rec}_i}(x, i, \vec{v})\), \(x\) will be renamed as \(u\), and \(i\) will not be renamed as they are program variables, but \(\vec{v}\) are existentially quantified and should be replaced with fresh variable names. Due to the restrictions on the recursive definitions, every \(\vec{v}\) is unique and can be determined by dereferencing \(u\) on the corresponding pointer fields, say \(\text{pf}_i^{\text{rec}_i}(x, i, \vec{v})\). Hence we can replace each \(v\) in \(\vec{v}\) distinctly as \(u^{\text{rec}_i}\). Let the renamed formula be \(\text{def}^{\text{rec}_i}(u, i, \vec{v})\), then we can derive

\[\text{Unfold}^{\text{rec}_i}(i, u) \equiv \left( \text{reach}^{\text{rec}_i}(u) = \text{def}^{\text{rec}_i}(u, i, \vec{v}) \right) \land (u \in R_i) \rightarrow \left( \left( \text{rec}_i(u) \leftrightarrow \text{def}^{\text{rec}_i}(u, i, \vec{v}) \right) \land \bigwedge_{i \leq n} (\text{pf}_i^{\text{rec}_i}(u) = u^{\text{rec}_i}) \right) \]

Now the footprint unfolding is just unfolding every dereferenced variable for every timestamp at the beginning and the end and the program, as well as before/after each procedure call, as long as the variable is allocated at the current timestamp. As a special location, \(\text{nil}\) is also unfolded with respect to each recursive definition and each timestamp. We can simply make the conjunction of all these unfoldings:

\[\text{Unfold} \equiv \bigwedge_{i \in [n]} \bigwedge_{\text{Def}(x)} \bigwedge_{\text{Field}(y)} \bigwedge_{i \leq n} \left( \text{Unfold}^{\text{rec}_i}(x, i, \vec{v}) \land \text{Unfold}^{\text{rec}_i}(y, i, \vec{v}) \right) \]

We use a predicate \(\text{fp}\) to indicate the footprint of the program. The fact that the footprint is exactly the set of locations pointed by
the dereferenced variables can be stated using a formula:

\[
\text{Footprint} \equiv \bigwedge_{w \in \text{Deref}} \left( \bigvee_{v} (u = v) \leftrightarrow \text{fp}(u) \right)
\]

We may also know that the recursive definitions or fields on a location is unchanged during a segment or procedure call of the program, because the location is not affected by the segment or procedure call. We first define a formula expressing that a recursive definition and its corresponding reach set on a location is unchanged between two timestamps:

\[
\text{UNCHANGED}^{\text{PF}}(u, 1, 1) \equiv \text{rec}_{c}(u) = \text{rec}_{c}'(u) \land \text{reach}_{c}^{\text{PF}}(u) = \text{reach}_{c}'^{\text{PF}}(u)
\]

In the \(d\)-th segment of the program, for each non-footprint location variable \(u\) and recursive predicate \(p'\) (or similarly for each recursive function \(f'\)), \(p_{j}(u)\) and \(\text{reach}_{j}^{u}(u)\) are unchanged, if the reach set \(\text{reach}_{j}^{u}(u)\) is not modified during this segment. This fact can be captures as follows:

\[
\text{SEGMENT UNCHANGED} \equiv \bigwedge_{0 \leq d \leq 2m} \bigwedge_{u \in \text{Deref}} \left( \neg \text{fp}(u) \rightarrow \left( \bigwedge_{w \in \text{Loc}} \left( \left( \text{reach}_{w}^{u}(u) \cap \text{Mod}_{d}(u) = \emptyset \right) \rightarrow \text{UNCHANGED}^{\text{PF}}(u, t_{j}, t_{j+1} - 1) \right) \right) \right)
\]

In the \(d\)-th procedure call of the program, for each variable \(u\), if \(u\) is not in the footprint, then for each recursive predicate \(\text{rec}_{c}'\), \(\text{rec}_{c}(u)\) and \(\text{reach}_{c}^{u}(u)\) are unchanged, if the reach set \(\text{reach}_{j}^{u}(u)\) is not affected during this call. Moreover, if \(u\) is in the footprint, then for each field \(pf\) (or \(df\)), \(\text{pf}_{i+1}(u)\) is unchanged if \(u\) itself is not affected during the call. This fact can be captures as follows:

\[
\text{CALLUNCHANGED} \equiv \bigwedge_{0 \leq d \leq 2m} \bigwedge_{u \in \text{Deref}} \left( \neg \text{fp}(u) \rightarrow \left( \bigwedge_{w \in \text{Loc}} \left( \left( \text{reach}_{w}^{u}(u) \cap \text{Call}_{d}(u) = \emptyset \right) \rightarrow \text{UNCHANGED}^{\text{PF}}(u, t_{j}, t_{j+1} - 1) \right) \right) \right) \land \left( \bigwedge_{w \in \text{Loc}} \left( pf_{j+1}(u) = pf_{j}(u) \land \bigwedge_{d} \left( df_{j+1}(u) = df_{j}(u) \right) \right) \right)
\]

We may also incorporate the fact that the reach set on every non-footprint location \(u\) contains itself:

\[
\text{SELFREACH} \equiv \bigwedge_{0 \leq d \leq 2m} \bigwedge_{u \in \text{Deref}} \left( u \neq \text{nil} \rightarrow \left( u \in \bigwedge_{w \in \text{Loc}} \left( \text{reach}_{w}^{u}(u) \cap \text{reach}_{w}^{u+1}(u) \right) \right) \right)
\]

Now we can strengthen the verification condition by incorporating all the derived formulas above:

\[
\psi_{\text{VC}} \equiv \psi_{\text{VC}} \land \text{UNFOLD} \land \text{FOOTPRINT} \land \text{SEGMENT UNCHANGED} \land \text{CALLUNCHANGED} \land \text{SELFREACH}
\]

Since the incorporated formulas are all satisfied when interpreted on a normal execution \(E\), we can reduce the validity of \(\psi_{\text{VC}}\) to the validity of \(\psi_{\text{VC}}\).

**Theorem 6.2.** Given a Hoare-triple \([\phi_{pre}, P, \psi_{post}]\), its verification condition \(\psi_{\text{VC}}\) is valid if and only if \(\psi_{\text{VC}}\) is valid.

### 6.4 Formula Abstraction

While the strengthened verification condition \(\psi_{\text{VC}}\) is still undecidable, as we argued before, it is often sufficient to prove it by assuming that the recursive definitions are arbitrary, or uninterpreted. Moreover, the uninterpreted formula falls in the array property fragment [12], whose satisfiability is decidable and is supported by modern SMT solvers such as Z3 [17].

To prove \(\psi_{\text{VC}}\), we first replace each recursive predicate \(\text{rec}_{j}\) with an uninterpreted predicate \(\text{rec}_{j}\), and replacing the corresponding reach set \(\text{reach}_{c}^{u}\) with an uninterpreted \(\text{reach}_{c}^{u}\). Let the result formula be \(\psi_{\text{APF}}\). This conversion is called formula abstraction, which is sound: if \(\psi_{\text{APF}}\) is valid, so is \(\psi_{\text{VC}}\). The formula abstraction is the gist of our natural proof scheme. When a proof for \(\psi_{\text{VC}}\) is found, we call it a natural proof for \(\psi_{\text{VC}}\).

**Remark:** Obviously the natural proof scheme is incomplete since the formula abstraction throws the semantics of the recursive definitions away and treats them as arbitrary. When the natural proof fails, there exists some spurious counterexample that satisfies \(\psi_{\text{VC}}\) but not \(\psi_{\text{APF}}\), as the counterexample interprets some recursive definition incorrectly. The incorrect interpretation may fall into one of the two following cases.

In the first case, the interpretation is not a fixed point of the propagation function with respect to the recursive definitions. In this case, the interpretation must be incorrect on some non-footprint locations, as we have precisely unfolded the recursive definitions on the footprint. The imprecisionness on non-footprint location is reasonably expected, and is not guaranteed in the natural proof scheme.

The second case is more subtle, which is when the interpretation is a non-least fixed point. In this case, the interpretation may still be incorrect even all locations are in the footprint. While we don’t have any specific mechanism to avoid this situation, in practice, the user may exclude this case by giving recursive definitions such that the fixed point is unique. First, it is noteworthy that we can define a recursive set-of-locations as an overapproximation of the reach set. Moreover, using this recursive function we can precisely describe whether a location is within a cycle or not. Secondly, typical recursive definitions recursively reduce the definition on a location to that of its neighbors, till \(\text{nil}\) is reached. For these definitions, we can modify them so that it is defined recursively on a location \(l\) only if \(l\) is not in a cycle (which is expressible in DRAAD), otherwise a default value is given. It is not hard to inductively prove that the fixed point is unique for the modified definitions.

Now our goal is to check the satisfiability of \(\neg \psi_{\text{APF}}\) in a decidable theory. Note that \(\neg \psi_{\text{APF}}\) is mostly expressible in the quantifier-free theory of arrays, maps, uninterpreted functions, and integers: \(\text{Loc}\) can be viewed as an uninterpreted sort; each pointer field \(pf\) can be viewed as an array with both indices and elements of sort \(\text{Loc}\); each data field \(df\) can be viewed as an array with indices of sort \(\text{Int}\) and elements of sort \(\text{Int}\) or \(\text{Mult}\). Moreover, each array update operation of the form \(\text{array}[\text{elem} \leftarrow \text{key}]\) can be viewed as a read-over-write operation in the array property fragment, and each set-operation (union, intersection, etc.) can be viewed as a mapping function maps a Boolean operation (\&\&, ||, etc.) on arrays.

The only construct in \(\neg \psi_{\text{APF}}\) that escapes the quantifier-free formulation is the \(\leq\) relation between integer sets/multisets; but this can be captured using the array-property fragment, which is decidable [12]. For each atomic formula of the form \(S_{1} \leq S_{2}\), if \(S_{1}\) and \(S_{2}\) are sets of integers, we can replace the formula with a universally quantified formula as follows:

\[
\forall i_{1}, i_{2}. \; (i_{1} < i_{2} \rightarrow (\neg S_{1}[i_{1}] \lor \neg S_{1}[i_{2}])))
\]

Similarly, if \(S_{1}\) and \(S_{2}\) are integer multisets, we can replace the formula with

\[
\forall i_{1}, i_{2}. \; (i_{1} < i_{2} \rightarrow (S_{1}[i_{1}] = 0 \lor S_{1}[i_{2}] = 0))
\]

We thus obtain a formula \(\psi_{\text{APF}}\) whose satisfiability is decidable.

**Theorem 6.3.** Given a Hoare-triple \([\phi_{pre}, P, \psi_{post}]\), if the derived array formula \(\psi_{\text{APF}}\) is satisfiable, then the Hoare-triple is valid.

\[\square\]
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>#Rec. Defs</th>
<th>Routine</th>
<th>#Basic Blocks</th>
<th>Total Time (s)</th>
<th>VC proved valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Examples</td>
<td>2</td>
<td>find</td>
<td>4</td>
<td>0.02</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f-insert</td>
<td>2</td>
<td>0.01</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b-insert</td>
<td>3</td>
<td>0.02</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete</td>
<td>5</td>
<td>0.05</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>copy</td>
<td>3</td>
<td>0.11</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>append</td>
<td>3</td>
<td>0.05</td>
<td>Yes</td>
</tr>
<tr>
<td>Sorted List</td>
<td>3</td>
<td>find</td>
<td>5</td>
<td>0.04</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>insert</td>
<td>4</td>
<td>0.08</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete</td>
<td>3</td>
<td>0.05</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>merge</td>
<td>6</td>
<td>1.68</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ins-sort</td>
<td>4</td>
<td>0.13</td>
<td>Yes</td>
</tr>
<tr>
<td>Doubly-Linked List</td>
<td>2</td>
<td>f-insert</td>
<td>2</td>
<td>0.04</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b-insert</td>
<td>3</td>
<td>0.29</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete</td>
<td>5</td>
<td>0.31</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>append</td>
<td>3</td>
<td>0.54</td>
<td>Yes</td>
</tr>
<tr>
<td>Cyclic List</td>
<td>1</td>
<td>f-insert</td>
<td>1</td>
<td>0.01</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b-insert</td>
<td>3</td>
<td>0.05</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f-delete</td>
<td>2</td>
<td>0.02</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b-delete</td>
<td>6</td>
<td>0.06</td>
<td>Yes</td>
</tr>
<tr>
<td>Max-Heap</td>
<td>2</td>
<td>heapify</td>
<td>16</td>
<td>10.95</td>
<td>Yes</td>
</tr>
<tr>
<td>BST</td>
<td>2</td>
<td>find</td>
<td>6</td>
<td>0.12</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>insert</td>
<td>5</td>
<td>0.53</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete</td>
<td>6</td>
<td>0.49</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rm-root</td>
<td>5</td>
<td>3.02</td>
<td>Yes</td>
</tr>
<tr>
<td>Treap</td>
<td>3</td>
<td>find</td>
<td>6</td>
<td>0.20</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>insert</td>
<td>7</td>
<td>89.87</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete</td>
<td>6</td>
<td>2.71</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rm-root</td>
<td>7</td>
<td>83.18</td>
<td>Yes</td>
</tr>
<tr>
<td>AVL-Tree</td>
<td>3</td>
<td>insert</td>
<td>7</td>
<td>10.07</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete</td>
<td>13</td>
<td>42.92</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>balance</td>
<td>6</td>
<td>6.76</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lefmost</td>
<td>3</td>
<td>1.17</td>
<td>Yes</td>
</tr>
<tr>
<td>RB-Tree</td>
<td>4</td>
<td>insert</td>
<td>13</td>
<td>1200.56</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>delete</td>
<td>19</td>
<td>45.85</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lefmost</td>
<td>3</td>
<td>0.51</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l-fix</td>
<td>5</td>
<td>19.21</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r-fix</td>
<td>5</td>
<td>23.91</td>
<td>Yes</td>
</tr>
<tr>
<td>Total</td>
<td>205</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. Results of program verification (see details at http://www.cs.illinois.edu/~qiu2/dryad)

7. Experimental Evaluation

We have implemented the natural proof methodology for Dryad presented in this paper as a prototype verifier. The verifier takes as input a set of user-defined recursive definitions, a set of procedure declarations with contracts, and a set of straight-line programs annotated with a pre-condition and a post-condition specifying a set of partial correctness properties including structural, data and separation requirements. Both the contracts and pre-/post-conditions are written in Dryad. Using the verifier, we successfully proved the partial correctness of 37 routines over a large class of data-structures including sorted lists, doubly-linked lists, cyclic lists and trees. Each routine is split into a number of straight-line programs (called basic blocks). For each basic block, the verifier automatically generates the abstracted formula $\phi_{\text{APP}}$ following the procedure described in Section 6, and passes $\phi_{\text{APP}}$ to Z3 [17], a state-of-the-art SMT solver, to check the satisfiability in the decidable theory of array property fragment.

The data-structure routines are given below; in general, the properties checked formalize the complete verification of the routine, capturing the precise structure of the resulting data-structure, the precise change to the data stored in the nodes, and the precise heaplet modified by the routine.

The set of Linux examples are taken from a Linux library of singly-linked lists. The routine $f\text{-insert}/b\text{-insert}$ inserts a key at the beginning/end of the list recursively. The routine copy clones a list and returns a new list with the same elements. The routines append, find, delete are implementations of the standard operations on singly-linked lists. The Sorted List examples include standard find, insert and delete routines implemented using recursion; the merge routine merges two sorted lists into a single sorted list; the ins-sort routine does insertion-sort of a given key, using the insert sub-routine.

The Doubly-Linked List suite includes the routines $f\text{-insert}/b\text{-insert}$ that inserts a key at the front/end of a doubly-linked list recursively, and delete recursively deletes all occurrences of a particular key. The routine append appends one list to another. We check if all these routines return a doubly-linked list with the keys in the list as expected. The Cyclic List routine $f\text{-insert}$ inserts a new node next to the head of the cyclic list, $b\text{-insert}$ traverses the list and inserts a new node just before the head of the list. Routines $f\text{-delete}$ and $b\text{-delete}$ do the delete operation similarly. In all these routines, we check if the list returned is a cyclic-list with appropriate keys.

The Max-Heap routine heapify accepts an almost max-heap in which the heap property is violated only at the root, both of whose children are max-heaps, and recursively descends the tree to restore the max-heap property. For Binary Search Trees, the find, insert and delete routines implement the standard operations recursively, and rm-root is an auxiliary routine called in delete. Treap data-structure routines include find, insert, delete and rm-root routines, similar to their BST counterparts. AVL Tree routines include the insert/delete routines, and as these routines return from the recursion they call balance to restore balance. The balance routine accepts an almost-AVL tree and checks the balancedness and performs one or two rotations to restore balance. For these routines, we prove that they return an AVL tree, that the multiset of keys is as expected, and that the height increases/decreases by at most 1. The leftmost routine is an auxiliary method that recursively finds the leftmost leaf and returns its key. We prove it returns the smallest key and all of the recursive definitions are preserved. Similarly, the Red-Black Tree routine insert recursively inserts a key into a red-black sub-tree, and performs several rotations and colorings to fix the red-black tree property. The delete routine deletes the key, possibly violating the red-black tree property, and as it returns from recursion, it calls the auxiliary left-fix or right-fix routine, which rotates and recolors the tree to restore the red-black tree property.

The experimental results are tabulated in Figure 9. For each data structure, we report the number of user-defined recursive definitions and the number of routines. For each routine, we report the number of basic blocks, the time taken by Z3 to determine satisfiability, and whether Z3 was able to prove the validity of the VCs.

We are encouraged by the fact that all these VCs that were generated by the natural proof methodology set forth in this paper were proved by Z3 efficiently. To the best of our knowledge, this is the first terminating automatic mechanism that can prove such a wide variety of data-structure algorithms full-functionally correct, involving such complex properties of structure, data and separation.

Note that our natural proof mechanism presented in [25] can only prove tree properties, and further, is restricted to extremely stringent conditions of pre- and post-conditions (for example, two trees cannot be given as parameters for a procedure, a procedure cannot modify the input node and return another tree, etc.). However, on the tree data-structure examples, the algorithm in [25] performs better as it has been honed to work only for trees, while graph algorithms on footprints check the tree property as opposed to the logical mechanism dealing with it. The technique in this paper is more general, and can handle arbitrary separation property expressed in Dryad, and is completely based on logic. The results in [25] do suggest that for certain structural properties, moving their...
check to a simpler graph algorithm outside of logic may be more efficient in practice.

It also worth mentioning that in the process of experiments, we did make some unintentional mistakes, in writing both the basic blocks and the annotations. For example, forgetting to free the deleted node, or using $\wedge$ instead of $\cdot$ in the specification between two disjoint heapsets, were common mistakes. In these cases, Z3 provided counter-examples to the verification condition that captured the essence of the bugs, and turned out to be very helpful for us to debug the specification. These debugging hints are usually not available in other incomplete proof systems.

The experimental results show that Divad is a very expressive logic that allows us to express natural and recursive properties of structure, data and separation, and the natural proofs often exist and can be found efficiently, in order to prove full-functional correctness of many data-structure manipulating algorithms.

(A) Proof of Theorem 6.1

Proof. We prove the soundness by contradiction. Assume the Hoare-triple $[\psi_{pre}] P [\psi_{post}]$ is not valid. Assume $P$ consists of $n$ statements, then there is an execution $E$, which can be represented as a state sequence $(C_0, \ldots, C_n)$ where each $C_i = (R_i, s_i, h_i)$, such that $(C_0, R_0)$ satisfies $\psi_{pre}[0], (C_n, R_n)$ satisfies $\psi_{post}[n]$, and the whole execution is memory free error. Then by the definitions of Pre, Post and NonNullDereference, and Theorem 5.1, $E \models Pre \And NonNullDereference \And Post$. Now it suffices to show that $E \models Monstration$, in which case $E$ dissatisfies $\psi_{vc}$. The contradiction will conclude the proof.

Since $Monstration \equiv \bigwedge_{i \in [n]} \psi_i$, we just need to prove $E \models \psi_i$ for each $i \in [n]$, by case analysis on the type of the $i$-statement in $P$.

[ $u \vdash v$ ]

$\psi_i \equiv u = v \And R_i = R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1)$

The variable assignment makes $u$ points to where $v$ points to. Hence $u = v$. Since the heap is unmodified from $C_{i-1}$ to $C_i$, the heap domain remains the same ($R_i = R_{i-1}$), and all the field functions remain the same ($FieldsUnmod(PF \cup DF, i, i-1)$).

[ $u \vdash \textit{nill}$ ]

$\psi_i \equiv u = \textit{nill} \And R_i = R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1)$

The variable assignment makes $u$ points to nill, so $u = \textit{nill}$. Similar to the above case, the heap is also unmodified from $C_{i-1}$ to $C_i$.

[ $u \vdash v, pf$ ]

$\psi_i \equiv v \in R_{i-1} \And u = pf_{i-1}(v) \And R_i = R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1)$

The dereferencing of $v$ implies that $v$ points to a valid location at timestamp $i-1$, i.e., $v \in R_{i-1}$. Moreover, the assignment makes $u$ points to the $pf$ field of $v$ at timestamp $i-1$, formally $u = pf_{i-1}(v)$. Similar to the above case, the heap is also unmodified from $C_{i-1}$ to $C_i$.

[ $u \vdash pf$ ]

$\psi_i \equiv \begin{cases} u = pf_{i-1}(v) & \text{if } v \in R_{i-1} \And R_i = R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1) \\ u \not\in R_{i-1} & \text{if } u \not\in R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1) \end{cases}$

The other field functions also remain the same, which is captured by $FieldsUnmod(PF \cup (DF \setminus \{pf\}), i, i-1)$.

[ $j := u.df$ ]

$\psi_i \equiv u \in R_{i-1} \And j = df_{i-1}(u) \And R_i = R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1)$

Similar to the $u \vdash v, pf$ case.

[ $u.df[j] := j$ ]

$\psi_i \equiv u \in R_{i-1} \And df[j] = df_{i-1}(u) \And R_i = R_{i-1} \And FieldsUnmod(PF \cup (DF \setminus \{df\}), i, i-1)$

Similar to the $u \vdash pf$ case.

[ $j := aexpr$ ]

$\psi_i \equiv j = aexpr \And R_i = R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1)$

The statement assigns the value of aexpr, which is expressible in our logic, to $j$. Hence $j = aexpr$. The rest is similar to other variable assignment cases.

[ $u \vdash \textit{new}$ ]

$\psi_i \equiv new_i \not\in \textit{nill} \And u = \textit{new} \And new_i \not\in R_{i-1} \And R_i = R_{i-1} \And \{new_i\} \And \bigwedge_{i \in [0]} (pf_i = pf_{i-1}(\textit{nill} \leftarrow new_i)) \And df_i = df_{i-1}(0 \leftarrow new_i))$

This statement makes $u$ points to a freshly allocated location, namely $new_i$, in $E$. So it is clear that $new_i \not\in \textit{nill} \And u = \textit{new}$. Since the heap domain at timestamp $i$ is an extension of that at timestamp $i-1$ by adding $new_i$, we know that $new_i \not\in R_{i-1} \And R_i = R_{i-1} \And \{new_i\}$. By default, for new, each pointer field initially points to nill, each data field initially stores $0$. The remaining portion of the heap is exactly the same as $C_{i-1}$. Hence $\bigwedge_{i \in [0]} (pf_i = pf_{i-1}(\textit{nill} \leftarrow new_i)) \And df_i = df_{i-1}(0 \leftarrow new_i))$

[ $\textit{free } u$ ]

$\psi_i \equiv u \in R_{i-1} \And R_i = R_{i-1} \setminus \{u\} \And FieldsUnmod(PF \cup DF, i, i-1)$

This statement removes the location pointed by $u$ from the heap. So the old heap contains this location, and the new heap can be obtained by subtracting it from the old heap: $u \in R_{i-1} \And R_i = R_{i-1} \setminus \{u\}$. Since the domain is shrunk, the field function can be simply unchanged.

[ $\textit{assume } aexpr$ ]

$\psi_i \equiv aexpr \And R_i = R_{i-1} \And FieldsUnmod(PF \cup DF, i, i-1)$

The assumed condition $aexpr$, which can be expressed in our logic, must be true. The heap is simply unmodified.

[ $u \vdash f(v, z)$ ]

$\psi_i \equiv T(\psi_{pre}(v, z, c_d), \text{Call}_d)[i-1] \And T(\psi_{post}(u, v, z, c_d), \text{Return}_d)[i] \And (R_{i-1} \setminus \text{Call}_d) \And \text{Return}_d = \emptyset \And R_i = (R_{i-1} \setminus \text{Call}_d) \And \text{Return}_d$

where $d$ is the index such that $td = i$

As assumed, this call is the $d$-th procedure call in $P$, and $C_{i-1}$ satisfies the associated precondition by the heaplet defined by $\text{Call}_d$, and $C_i$ satisfies the associated postcondition by the heaplet defined by $\text{Return}_d$. Then formally we have $T(\psi_{pre}(v, z, c_d), \text{Call}_d)[i-1] \And T(\psi_{post}(u, v, z, c_d), \text{Return}_d)[i]$ being satisfied by $E$.

Due to the framing property of the separation semantics, the portion of $C_{i-1}$ that is not required by $\psi_{pre}$ remains unchanged, and is disjoint from $\text{Return}_d$ (since the returned location is assigned to $u$, the variable $\text{ret loc}$ can be replaced with $u$). This property can be expressed as $(R_{i-1} \setminus \text{Call}_d) \And \text{Return}_d = \emptyset \And R_i = (R_{i-1} \setminus \text{Call}_d) \And \text{Return}_d$.

[ $j := g(v, z)$ ]

$\psi_i$ is defined in the same way as the above case, except replacing $u$ with $j$. 

2012/8/11
The proof is also similar to the above case.

\[ \square \]

References


