Natural Proofs for Structure, Data, and Separation

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

Abstract

We propose *natural proofs* for reasoning with programs that manipulate data-structures against specifications that describe the structure of the heap, the data stored within it, and separation and framing of sub-structures. Natural proofs are a subclass of proofs that are amenable to completely automated reasoning, that provide sound but incomplete procedures, and that capture common reasoning tactics in program verification. We develop a dialect of separation logic over heaps, called DRYAD, with recursive definitions that avoids explicit quantification. We develop ways to reason with heaplets using classical logic over the theory of sets, and develop natural proofs for reasoning using proof tactics involving disciplined unfoldings and formula abstractions. Natural proofs are encoded into decidable theories of first-order logic so as to be discharged using SMT solvers.

We also implement the technique and show that a large class of more than 100 correct programs that manipulate data-structures are amenable to full functional correctness using the proposed natural proof method. These programs are drawn from a variety of sources including standard data-structures, the Schorr-Waite algorithm for garbage collection, a large number of low-level C routines from the Glib library and OpenBSD library, the Linux kernel, and routines from a secure verified OS-browser project. Our work is the first that we know of that can handle such a wide range of full functional verification properties of heaps automatically, given pre/post and loop invariant annotations. We believe that this work paves the way for deductive verification technology to be used by programmers who do not (and need not) understand the internals of the underlying logic solvers, significantly increasing their applicability in building reliable systems.

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

Keywords heap analysis; data structures; natural proofs; separation logic; SMT solvers

1. Introduction

In recent years, the *automated deductive verification paradigm* for software verification that combines user written modular contracts and loop invariants with automated theorem proving of the resulting verification conditions has become very powerful. The latter process is often executed by automated logical decision procedures supported by SMT solvers, which have emerged as robust and powerful engines to automatically find proofs. Several techniques and tools have been developed [2, 16, 20] and there have been several success stories of large software verification projects using this approach (the Verve OS project [42], the Microsoft hypervisor verification project using VCC [16], and a recent verified-for-security OS+browser for mobile applications [29], to name a few).

Verification conditions do not, however, always fall into decidable theories. In particular, the verification of properties of the *dynamically modified heap* is a 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 to a doubly-linked list, or to 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 one list and not the other and leaves the two lists disjoint at exit, etc.).

The fact that the dynamic heap contains an unbounded number of locations means that expressing the above properties requires *quantification* in some form, which immediately precludes the use of most SMT decidable theories (there are only a few of them known that can handle quantification; e.g., the array property fragment [12] and the STRAND logic [25, 26]). Consequently, *expressing* such properties naturally and succinctly in a logical formalism has been challenging, and reasoning with them automatically even more so.

For instance, in the Boogie line of tools (including VCC) of writing specifications using first-order logic and employing SMT solvers to validate verification conditions, the specification of invariants of even simple methods like *singly-linked-list insert* is tedious. In such code[1], second-order properties (reachability, acyclicity, separation, etc.) are smuggled in using carefully chosen *ghost variables*; for example, acyclicity of a list is encoded by assigning a ghost number (idx) to each node in the list, with the property that the numbers associated with adjacent nodes strictly increase going down the list. These ghost variables require careful manipulation when the structures are updated; for example, inserting a node may require updating the ghost numbers for other nodes in the list, in order to maintain the acyclicity property. Once such a ghost-encoding of the specification is formulated, the validation of verification conditions, which typically have quantifiers, are dealt with using sound heuristics (a wide variety of them including e-matching, model-based quantifier instantiation, etc. are available), but are still often not enough and have to be augmented by *instantiation triggers* from the verification engineer to help the proof go through.

[1] http://vcc.codeplex.com/SourceControl/changeset/view/dcaa4d0ee8c2#vcc/Docs/Tutorial/c/7.2.list.c

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

In recent years, *separation logic*, especially in combination with recursive definitions, has emerged as a much more succinct and natural logic to express properties about structure and separation [32, 36]. However, the validation of verification conditions resulting from separation logic invariants are also complex, and has eluded automatic reasoning and exploitation of SMT solvers (even more so than tools such as Boogie that use classical logic). Again, help from the user in proving the verification conditions are currently necessary— the tools VERIFAST [20] and BEDROCK [15], for instance, admit separation logic specifications but require the user to write low-level lemmas and proof tactics to guide the verification. For example, in verifying an in-place reversal of a linked list[2], BEDROCK would require several lemmas and a hint package be supplied at the level of the code in order for the proof to go through.

The work in this paper is motivated by the opinion that entirely decidable logics are too restrictive, in general, to support the verification of complex specifications of functional correctness for heap manipulating programs, and the other extreme of user-supplied proof tactics and lemmas is too tedious, requiring of the user too much knowledge of the underlying proof systems/decision procedures. Our aim is to build completely automatic, sound, but incomplete proof techniques that can solve a large class of properties involving complex data-structures.

The natural proof methodology:
Our proof methodology of natural proofs was first proposed in a paper by Madhusudan et al on natural proofs for tree data-structures last year at POPL [27]. Natural proofs exploit a *fixed* set of proof tactics, keeping the expressiveness of powerful logics, retaining the automated nature of proving validity, but giving up on completeness (i.e., giving up decidability, retaining soundness). The idea of natural proofs [27] 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 even 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. The natural proofs developed in [27] were too restrictive, however, handling only *single* trees, with no scope for handling multiple or more complex data-structures and their separation (see section on Related Work for more details).

The aim of this paper is to provide natural proofs for general properties of structure, data, and separation. Our contributions are: (a) we propose DRYAD, a dialect of separation logic for heaps, with no explicit (classical) quantification but with recursive definitions, to express second-order properties; (b) show that DRYAD is both powerful in terms of expressiveness, and that the strongest-post of DRYAD specifications with respect to bounded code segments can be formulated in DRYAD, (c) show how DRYAD has been designed so that it can be systematically converted to *classical* logic using the theory of sets, allowing us to connect the more natural and succinct specifications to more verbose but classical logic, and (d) develop a natural proof mechanism for classical logics with recursion and sets that implement sound but incomplete reductions to decidable theories that can be handled by an SMT solver.

DRYAD*: A separation logic with determined heaplets*
The primary design principle behind separation logic is the decision to express *strict specifications*— logical formulas must naturally 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 procedure, implicitly leaving the rest of the heap and its properties unchanged across a call to this 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).

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 are quite complex and not easily amenable to automated verification. 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:

$$tree(x) :: (x = \mathtt{nil} \wedge \mathtt{emp}) \vee (x \longmapsto (l, r) * tree(l) * tree(r))$$

We first define a new logic, DRYAD, that permits no explicit quantification, but permits powerful recursive definitions to define integers, sets/multisets of integers, and sets of locations, using least fixed-points. The logic DRYAD furthermore has a heaplet semantics and allows the spatial conjunction operator $*$. However, a key design feature of DRYAD is that the heaplet for recursive formulas is essentially *determined* by the syntax as opposed to the semantics. In classical separation logic, a formula of the form $\alpha * \beta$ says that the heaplet can be partitioned into *any* two disjoint heaplets, one satisfying α and the other β. In DRYAD, the heaplet for a (complex) formula is *determined* and hence if there is a way to partition the heaplet, there is precisely *one* way to do so. We have found that most uses of separation logic to express properties can be written quite succinctly and easily using DRYAD (in fact, it is *easier* to write such deterministic-heap specifications). The key advantage is that this eliminates implicit existential quantification the separation operator provides. In a verification condition that combines the pre-condition in the negative and the post-condition in the positive, the classical semantics for separation logic invariably introduces universal quantification in the satisfiability query for the negation of the verification condition, which in turn is extremely hard to handle.

In DRYAD, the semantics of a recursive definition $r(x)$ (such as *tree* above), requires that the heaplet be determined and defined as the set of all locations reachable from the node x through a set of pointer-fields $f_1, \ldots, f_k$ without passing through a set of locations (given by a set of location terms $t_1, \ldots t_n$). While our logical mechanisms can be extended beyond this notion (in deterministic ways), we have found that this captures most common properties required in proving data-structure manipulating programs correct.

Translating* DRYAD *to classical logic with recursion:
The second key step in our paradigm is a technique to bridge the gap from separation logic to classical logic in order to utilize efficient decision procedures supported by SMT solvers. We show that heaplet semantics and separation logic constructs of DRYAD can be effectively translated to classical logic where heaplets are modeled as *sets of locations*. We show that DRYAD 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 (the recursion-free formulas are in a decidable theory). The carefully designed DRYAD logic with determined heaplet semantics ensures that there is no quantification in the resulting formula in classical logic. The heaplets of recursively defined prop-

[2] http://plv.csail.mit.edu/bedrock/Tutorial.html

erties, which are defined using the set of all *reachable* nodes, are translated to recursively defined sets of locations.

Natural proofs for* DRYAD*:
Finally, we develop a natural proof methodology for DRYAD by showing a natural proof mechanism for the equivalent formulas in classical logic. The basic proof tactic that we follow is not just dependent on the formula embodying the verification condition, but also on the precise footprint touched by the program segment being verified. We unfold recursive definitions precisely across footprints, translating them to the frontier of the footprint, and then use a form of *formula abstraction* that treats recursive formulas on frontier nodes as uninterpreted functions. The resulting formula falls in a logic over sets and integers, which is then decided using the theory of uninterpreted functions and arrays using SMT solvers. The key feature is that heaplets and separation logic constructs, which get translated to recursively defined sets of locations, are unfolded along with other user-defined recursive definitions and formula-abstracted using this uniform natural proof strategy,

While our proof strategy is roughly as above, there are many technical details that are complex. 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. Also, 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 admits automatic decision procedures.

Implementation and Evaluation:
Our proof mechanisms are essentially a class of decidable proof tactics that result in sound but incomplete validation procedures. To show that this class of natural proofs is effective in practice, we provide a prototype implementation of our technique, which handles a low-level programming language with pre-conditions and post-conditions written in DRYAD. We show, using a large class of correct programs manipulating lists, trees, cyclic lists, and doubly linked lists as well as *multiple* data-structures of these kinds, that the natural proof mechanism succeeds in proving the verifications conditions automatically. These programs are drawn from a range of sources, from textbook data-structure routines (binary search trees, red-black trees, etc.) to routines from Glib low-level C-routines used in GTK+/Gnome to implement file-systems, from the Schorr-Waite garbage collection algorithm, to several programs from a recent secure framework developed for mobile applications [29]. Our work is by far the only one that we know of that can handle such a large class of programs, completely automatically. Our experience has been that the user-provided contracts and invariants are easily expressible in DRYAD, and the automatic natural proof mechanisms work extremely fast. In fact, contrary to our own expectations, we also found that the tool is useful in *debugging*: in several cases, when the annotations supplied were incorrect, the model provided by the SMT solver for the natural proof was useful in detecting errors and correcting the invariants/program.

2. Related Work

The natural proof methodology was introduced in [27] (see also [39]), but was exclusively built for tree data-structures. In particular, this work could only handle *recursive* programs, i.e., no while-loops, and even for tree data-structures, imposed a large number of restrictions on pre/post conditions for methods— the input to a procedure had to be only a single tree, the method can only return a single tree, and even then must havoc the input tree given to it. The lack of handling of multiple structures means that even simple programs like mergesort (that merges two lists), cannot be handled, and simple programs that manipulate two lists or two trees cannot be reasoned with. Also, structures such as doubly-linked lists, trees with parent pointers, etc. are out of scope of this work. Technically, in our present work, we can handle user-defined structures expressible in separation logic, multiple structures and their separation, programs with while-loops, etc., because of our *logical* treatment of separation logic using classical logic.

There is a rich literature on analysis of heaps in software. We omit discussing literature on general interactive theorem provers (like ISABELLE [31]) that require considerable manual guidance. We also omit a lot of work on analyzing shape properties of the heap [6, 13, 18, 28, 41], as they do not handle complex functional properties.

There are several proof systems and assistants for separation logic [32, 36] 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 [20], on the other hand, chooses flexibility of writing richer specifications over complete automation, but requires the user to provide inductive lemmas and proof tactics to aid verification. Similarly, BEDROCK [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. The idea of using regions (sets of locations) for describing heaps in our work also extends to describing frames for function calls, and the use for the latter is similar to implicit dynamic frames [38] in the literature. The crucial difference in our framework is that the implicit dynamic frames are syntactically determined, and amenable to quantifier-free reasoning. 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 mechanism; in fact, the resulting formulas are quantified and incompatible 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 [37], 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 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. DAFNY [23] and VCC [16] compile to BOOGIE [2] and generate VCs that are passed to SMT solvers. This approach requires significant ghost annotations, and annotations that explicitly express and manipulate frames. The JAHOB system [43, 44] 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 [21], or even interactive theorem provers such as ISABELLE [31] in the worst case. The success of the proof search also relies on users' manual guidance.

```
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 =/= nil)
    if (s.key > x.key) {
      t := s.key;
      s.key := x.key;
      x.key := t;
      heapify(s);
    }
}
```

$$\varphi_{pre} \equiv \Big(x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * mheap^{\Delta}_{\overrightarrow{pf}}(l) * mheap^{\Delta}_{\overrightarrow{pf}}(r)\Big) \land keys^{\Delta}_{\overrightarrow{pf}}(x) = K$$

$$\varphi_{post} \equiv mheap^{\Delta}_{\overrightarrow{pf}}(x) \land keys^{\Delta}_{\overrightarrow{pf}}(x) = K$$

`assume` $x.left_0 \neq nil$
`assume` $x.right_0 \neq nil$
$lx := x.left_0$
$rx := x.right_0$
`assume` $lx.key_0 < rx.key_0$
$s := x.right_0$
`assume` $s \neq nil$
`assume` $s.key_0 > x.key_0$
$t := s.key_0$
$s.key_1 := x.key_0$
$x.key_2 := t$
`heapify`(s)

$$\boldsymbol{mheap}^{\Delta}_{\overrightarrow{pf}}(x) \stackrel{def}{=} \Big(x = \mathtt{nil} \land \mathtt{emp} \lor (x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * (mheap^{\Delta}_{\overrightarrow{pf}}(l) \land \{k\} \geq keys^{\Delta}_{\overrightarrow{pf}}(l)) * (mheap^{\Delta}_{\overrightarrow{pf}}(r) \land \{k\} \geq keys^{\Delta}_{\overrightarrow{pf}}(r)))\Big)$$

$$\boldsymbol{keys}^{\Delta}_{\overrightarrow{pf}}(x) \stackrel{def}{=} \Big(x = \mathtt{nil} \land \mathtt{emp} : \emptyset\ ;\ x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * \mathtt{true} : keys^{\Delta}_{\overrightarrow{pf}}(l) \cup \{k\} \cup keys^{\Delta}_{\overrightarrow{pf}}(r)\ ;\ \text{default} : \emptyset\Big)$$

Figure 1. Motivating example: Heapify

The idea of unfolding recursive definitions and formula abstraction also features in the work by Suter et al. [39, 40], 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 Lisbq [22] and CSL [9, 10] offer such reasoning with restricted reachability predicates and quantification; see also the logics in [1, 7, 30, 33–35]. Strand is a relatively expressive logic that can handle some data-structure properties (like BSTs) and admits decidable fragments [25, 26], but is again not expressive enough for more complex properties of inductive data-structures. None of these logics can express the class of VCs for full functional verification explored in this paper.

3. Motivating Example

In this section we give intuition into our verification approach through a motivating example. Recall that a max-heap is a binary tree such that for each node n the key stored at n is greater than or equal to the keys stored at each of its children. Heaps are often used to implement priority queues. In Figure 1, in the lower right corner, we express the property that a location x points to a max-heap using recursive definitions $keys^{\Delta}_{\overrightarrow{pf}}(x)$ and $mheap^{\Delta}_{\overrightarrow{pf}}(x)$, with $\overrightarrow{pf} \equiv \{left, right\}$. These recursive definitions are written in Dryad, which is formally introduced in Section 4. Intuitively, Dryad extends quantifier free separation logic [32, 36] with recursive predicates and functions. These recursive definitions allow us to express structural and data properties on the heap, like those of max-heap, without explicit quantification.

For a location x, the recursive definition $keys^{\Delta}_{\overrightarrow{pf}}(x)$ returns the set of keys at the nodes of the tree rooted at x: if x is `nil` and the heaplet is empty, then the empty-set; otherwise, the union of the key stored at x and the keys stored in the left and right subtrees of x. Similarly, the recursive definition $mheap^{\Delta}_{\overrightarrow{pf}}(x)$ states that x points to a max-heap if: x is `nil` and the heaplet is empty; or x and the heaplets of the left and right subtrees of x are mutually disjoint (x points to a tree) and the key at x is greater than or equal to the keys of the left and right subtrees of x.

The method `heapify` in Figure 1 is at the heart of the procedure for deleting the root from a max-heap (removing the node with the maximum priority). If the max-heap property is violated at a node x while satisfied by its descendants, then `heapify` restores the max-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 φ_{pre} binds the free variable K to the set of keys of x. The postcondition states that after the procedure call, x satisfies the max-heap property and the set of keys of x is unchanged (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 Dryad formulas, like those in Figure 1, to formulas in classical logic with recursive predicates and functions is formally presented in Section 5. Intuitively, we associate a set of locations to each (spatial) atomic formula, 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 $x \mapsto \ldots$ is the singleton $\{x\}$; for recursive definitions like $mheap^{\Delta}_{\overrightarrow{pf}}(x)$ and $keys^{\Delta}_{\overrightarrow{pf}}(x)$, the domain of the heaplet is $reach_{\{left,right\}}(x)$, which intuitively is the set of locations reachable from x using the pointer fields *left* and *right*, and can be defined recursively.

As shown in Figure 1, φ_{pre} is a conjunction of two formulas. If G_{pre} is the domain of the heaplet associated to φ_{pre}, then the first conjunct requires G_{pre} to be the disjoint union of the sets $\{x\}$, $reach_{\{left,right\}}(left(x))$ and $reach_{\{left,right\}}(right(x))$. The second conjunct requires $G_{pre} = reach_{\{left,right\}}(x)$. From these heaplet constraints, we can translate φ_{pre} to the following formula in classical logic *over the global heap*:

$$\begin{aligned}
& G_{pre} = \{x\} \cup reach_{\{left,right\}}(left(x)) \cup reach_{\{left,right\}}(right(x)) \\
\land\ & x \notin reach_{\{left,right\}}(left(x)) \land x \notin reach_{\{left,right\}}(right(x)) \\
\land\ & reach_{\{left,right\}}(left(x)) \cap reach_{\{left,right\}}(right(x)) = \emptyset \land x \neq nil \\
\land\ & mheap(left(x)) \land mheap(right(x)) \\
\land\ & G_{pre} = reach_{\{left,right\}}(x) \land keys(x) = K
\end{aligned}$$

Similarly, we translate φ_{post} to

$$G_{post} = reach_{\{left,right\}}(x) \land mheap(x) \land keys(x) = K$$

Note that the recursive definitions *mheap* and *keys* without the "Δ" superscript are in the classical logic (without the heaplet constraint). Hence the recursive predicate *mheap* satisfies

$$\begin{aligned}
mheap(x) \leftrightarrow\ & x = nil \land reach_{\{left,right\}}(x) = \emptyset) \\
\lor\ & \big(x \neq nil \land x \notin reach_{\{left,right\}}(left(x)) \land x \notin reach_{\{left,right\}}(right(x)) \\
& \land reach_{\{left,right\}}(left(x)) \cap reach_{\{left,right\}}(right(x)) = \emptyset \\
& \land (reach_{\{left,right\}}(x) = \{x\} \cup reach_{\{left,right\}}(left(x)) \\
& \quad \cup reach_{\{left,right\}}(right(x))) \\
& \land mheap(left(x)) \land \{key(x)\} \geq keys(left(x)) \\
& \land mheap(right(x)) \land \{key(x)\} \geq keys(right(x))\big)
\end{aligned}$$

The right side of Figure 1 presents a basic path from method `heapify`, corresponding to the case when both children of x are not *nil* and the key of the right child is greater than the keys of the left child and the root. The subscript of a pointer/data field denotes the timestamp. A key insight is that any basic path touches a finite number of locations and may call some recursive procedures. We refer to the touched locations as the *footprint*, and to the adjacent locations which are not part of the footprint as the *frontier*. For this example, the footprint is $\{\ x,\ lx,\ rx\ \}$ (s is known to be equal with rx) and the frontier is $\{\ left_0(lx),\ right_0(lx),\ left_0(rx),\ right_0(rx)\ \}$. We capture the effect of the path until the call to `heapify` by

$$\begin{aligned} & left_0(x) \neq nil \wedge right_0(x) \neq nil \wedge lx = left_0(x) \wedge rx = right_0(x) \\ \wedge\ & key_0(lx) < key_0(rx) \wedge\ s = right_0(x) \wedge s \neq nil \\ \wedge\ & key_0(s) > key_0(x) \wedge t = key_0(s) \\ \wedge\ & key_1 = key_0\{s \leftarrow key_0(x)\} \wedge key_2 = key_1\{x \leftarrow t\} \end{aligned}$$

Once we have expressed the verification condition in classical logic with recursive definitions over the global heap, we prove it using the *natural proof* methodology. We unfold the recursive definitions $mheap(x)$, $keys(x)$ and $reach_{\{left,right\}}(x)$ for x, lx and rx (the footprint), thus evaluating them in terms of their values on the frontier. The call to `heapify` preserves the recursive definitions on locations reachable from lx, and modifies those on rx according to the pre/post condition. Finally, we abstract the recursive definitions on the frontier with uninterpreted functions. We decide the resulted formula (which is in a decidable logic) using an SMT solver. Section 6 describes this process in detail.

4. The Logic Dryad

In this section we present our logic Dryad$_{sep}$; this redefines the logic Dryad [27] 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. However, for brevity, we will refer to the new logic we propose as Dryad, and refer to the logic in [27] as Dryad$_{tree}$.

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, and that address arithmetic is precluded. We will use the term *locations* hence to refer to these records. We assume that every field is defined at every location, i.e., all memory records have the same layout (to simplify the presentation); our logic can easily be extended with record types.

Let $Bool = \{\texttt{true}, \texttt{false}\}$ 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 all finite subsets of A, and let $\mathcal{MS}(A)$ 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 recursively defined notions for a location to express second-order properties, denoted as a function $r : Loc \to D$. The codomain D can be Int_L, $\mathcal{S}(Loc)$, $\mathcal{S}(Int)$, $\mathcal{MS}(Int)_L$ or $Bool$, where Int_L and $\mathcal{MS}(Int)_L$ extend Int and $\mathcal{MS}(Int)$ to lattice domains, respectively, in order to give least fixed-point semantics (explained later in this section). Typical examples of these recursive definitions include the definitions of 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 φ is quantifier-free, but parameterized by a set of recursive definitions Def^{Δ}. The syntax of Dryad logic is given in Figure 2, where the syntax of formulas is followed by the syntax for recursive definitions. Most symbols in Dryad are common and self-explanatory. Note that the inequality ($<$ or $\leq$) between integer sets/multisets indicates that any integer in the left-hand side is less-than/not-greater-than any integer in the right-hand side. It is also noteworthy that the separating conjunction ($*$) from separation logic is also allowed, but only if it is not above any negation ($\neg$). We require that every recursive function/predicate used in the formula φ has a unique definition in Def^{Δ}. Each recursive function is parameterized by a set of pointer fields $\overrightarrow{pf}$ and a set of program variables $\vec{v}$, denoted as $f^{\Delta}_{\overrightarrow{pf},\vec{v}}$. The subscripts are used in defining the semantics of recursive functions in Section 4.2. We usually simply use f^{Δ} when the subscripts are not relevant in the context. Similarly, recursive predicates are denoted as $p^{\Delta}_{\overrightarrow{pf},\vec{v}}$ or simply p^{Δ}. The recursive functions are defined using the syntax:

$$(\varphi^f_1(x,\vec{v},\vec{s}) : t^f_1(x,\vec{s})\ ;\ \ldots\ ;\ \varphi^f_k(x,\vec{v},\vec{s}) : t^f_k(x,\vec{s})\ ;\ \text{default} : t^f_{k+1}(x,\vec{s}))$$

where $\varphi^f_u(x,\vec{v},\vec{s})/t^f_u(x,\vec{s})$ is a formula/term in our logic with $\vec{s}$ implicitly existentially quantified. The recursively defined predicates are defined using the syntax: $\varphi^p(x,\vec{v},\vec{s})$, which is a formula in our logic with $\vec{s}$ implicitly existentially quantified. The recursive function syntax above expresses a case-split, with the function evaluating to the first term whose guard evaluates to true. The restrictions on the recursive definitions are:

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

For examples of recursive functions and predicates, see the definitions $keys^{\Delta}_{\overrightarrow{pf}}(x)$ and $mheap^{\Delta}_{\overrightarrow{pf}}(x)$ in Figure 1, respectively. The set of program variables $\vec{v}$ parameterizing the definitions is empty in both these definitions and the set of implicitly existentially-quantified variables $\vec{s}$ is $\{k, l, r\}$.

4.2 Semantics

Our logic is interpreted on models that are *program states*:

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

- *$R \subseteq Loc \setminus \{nil\}$ is a finite set of locations;*
- *$s : Vars \to Int \cup Loc$ is a store mapping program variables to locations or integers (of appropriate type);*
- *$h : R \times (PF \cup DF) \to Int \cup Loc$ is a heaplet 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 heaplet h gives interpretations for pointer and data-fields only for elements in R.

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

A term/formula with free variables F is interpreted by interpreting the free variables in F using the map s from variables to values. The semantics of Dryad is similar to that of classical Separation Logic (*SL*). In particular, a term/formula without recursive definitions is interpreted exactly in the same way in Dryad and *SL*. Hence we first give the semantics of the non-recursive part, followed by the semantics of recursive definitions.

$$\begin{array}{lllll}
i^{\Delta}: Loc \to Int_L & sl^{\Delta}: Loc \to \mathcal{S}(Loc) & si^{\Delta}: Loc \to \mathcal{S}(Int) & msi^{\Delta}: Loc \to \mathcal{MS}(Int)_L & p^{\Delta}: Loc \to Bool \\
j \in Int_L \text{ Variables} & L \in \mathcal{S}(Loc) \text{ Variables} & S \in \mathcal{S}(Int) \text{ Variables} & MS \in \mathcal{MS}(Int)_L \text{ Variables} & q \in Bool \text{ Variables} \\
x \in Loc \text{ Variables} & c : Int_L \text{ Constant} & pf \in PF & df \in DF &
\end{array}$$

$$\begin{array}{rrcl}
Loc \text{ Term:} & lt & ::= & x \mid \mathtt{nil} \\
Int_L \text{ Term:} & it & ::= & c \mid j \mid i^{\Delta}_{\overrightarrow{pf},\vec{v}}(lt) \mid it + it \mid it - it \\
\mathcal{S}(Loc) \text{ Term:} & slt & ::= & \emptyset_l \mid L \mid \{lt\} \mid sl^{\Delta}_{\overrightarrow{pf},\vec{v}}(lt) \mid slt \cup slt \mid slt \cap slt \mid slt \setminus slt \\
\mathcal{S}(Int) \text{ Term:} & sit & ::= & \emptyset_s \mid S \mid \{it\} \mid si^{\Delta}_{\overrightarrow{pf},\vec{v}}(lt) \mid sit \cup sit \mid sit \cap sit \mid sit \setminus sit \\
\mathcal{MS}(Int)_L \text{ Term:} & msit & ::= & \emptyset_m \mid MS \mid \{it\}_m \mid msi^{\Delta}_{\overrightarrow{pf},\vec{v}}(lt) \mid msit \cup msit \mid msit \cap msit \mid msit \setminus msit
\end{array}$$

$$\begin{array}{rrcl}
\text{Positive Formula:} & \varphi & ::= & \mathtt{true} \mid \mathtt{false} \mid q \mid p^{\Delta}_{\overrightarrow{pf},\vec{v}}(lt) \mid \mathtt{emp} \mid lt \overset{\overrightarrow{pf},\overrightarrow{df}}{\longmapsto} (\vec{lt}, \vec{it}) \mid lt = lt \mid lt \neq lt \mid it \leq it \mid it < it \mid sit \leq sit \mid sit < sit \mid msit \leq msit \mid msit < msit \\
& & & \mid slt \subseteq slt \mid slt \not\subseteq slt \mid sit \subseteq sit \mid sit \not\subseteq sit \mid msit \sqsubseteq msit \mid msit \not\sqsubseteq msit \mid lt \in slt \mid lt \notin slt \mid it \in sit \mid it \notin sit \mid it \in msit \mid it \notin msit \\
& & & \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi \\
\text{Formula:} & \psi & ::= & \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi
\end{array}$$

$$\begin{array}{rcl}
\text{Recursive function}: \ f^{\Delta}_{\overrightarrow{pf},\vec{v}}(x) & \overset{def}{=} & (\varphi^f_1(x,\vec{v},\vec{s}) : t^f_1(x,\vec{s}) \ ; \ \ldots \ ; \ \varphi^f_k(x,\vec{v},\vec{s}) : t^f_k(x,\vec{s}) \ ; \ \text{default} : t^f_{k+1}(x,\vec{s})) \\
\text{Recursive predicate}: \ p^{\Delta}_{\overrightarrow{pf},\vec{v}}(x) & \overset{def}{=} & \varphi^p(x,\vec{v},\vec{s})
\end{array}$$

Figure 2. Syntax of Dryad

Before defining the semantics of formulas, we define the *pure* property for terms/formulas. Intuitively, a term/formula is pure if it is independent of the heap. Syntactically, a term/formula is pure if it does not contain `emp`, $\longmapsto$ or any recursive definition. Note that in *SL* all terms are pure, but in Dryad, a term can be impure if it contains a recursive function f^{Δ}.

Semantics of terms

Each $\mathcal{T}$-term evaluates to either a normal value of type $\mathcal{T}$, or to `undef`, which is only used in interpreting recursive functions (will be explained later). As a special value, `undef` will be propagated throughout the formula: if a formula φ contains a sub-term that evaluates to `undef`, then φ will evaluate to `false` if it appears positively, and will evaluate to `true` otherwise. Intuitively, `undef` cannot help in making the formula true over a model.

The *Loc* terms are evaluated as follows:

$$\begin{array}{rcl} [\![x]\!]_C & = & s(x) \\ [\![\mathtt{nil}]\!]_C & = & nil \end{array}$$

For any binary operator op, t op t' is evaluated as follows:

$$[\![t \text{ op } t']\!]_C = \begin{cases} [\![t]\!]_C \text{ op } [\![t']\!]_C & \text{if } t \text{ or } t' \text{ is pure} \\ [\![t]\!]_{C|R_1} \text{ op } [\![t']\!]_{C|R_2} & \text{else if there exist } R_1, R_2 \text{ such that} \\ & \quad R = R_1 \cup R_2, [\![t]\!]_{C|R_1} \neq \mathtt{undef} \\ & \quad \text{and } [\![t']\!]_{C|R_2} \neq \mathtt{undef} \\ \mathtt{undef} & \text{otherwise} \end{cases}$$

where op is interpreted in the natural way.

For singletons, $\{it\}$ will evaluate to $\emptyset$ if it evaluates to $-\infty$ or ∞:

$$[\![\{it\}]\!]_C = \begin{cases} \mathtt{undef} & \text{if } [\![it]\!]_C = \mathtt{undef} \\ \emptyset & \text{if } [\![it]\!]_C = -\infty \text{ or } \infty \\ \{[\![it]\!]_C\} & \text{otherwise} \end{cases}$$

$\{it\}_m$ and $\{lt\}$ evaluate similarly.

Semantics of formulas

The formula `true` is always interpreted to be *true*:

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

The formula `emp` asserts that the heap is empty:

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

The formula $lt \overset{\overrightarrow{pf},\overrightarrow{df}}{\longmapsto} (\vec{lt}, \vec{it})$ asserts that the heap contains exactly one record consisting of fields $\overrightarrow{pf}$ and $\overrightarrow{df}$, at address lt, with values $\vec{lt}$ and $\vec{it}$, respectively. Formally, the semantics of this formula is given as:

$$\begin{array}{rl} (R, s, h) \models lt \overset{\overrightarrow{pf},\overrightarrow{df}}{\longmapsto} (\vec{lt}, \vec{it}) \quad \text{iff} & R = \{[\![lt]\!]_{R,s,h}\} \text{ and} \\ & h([\![lt]\!]_{R,s,h}, pf_i) = [\![lt_i]\!]_{R,s,h} \quad \text{for corresponding } pf_i \text{ and } lt_i, \\ & h([\![lt]\!]_{R,s,h}, df_i) = [\![it_i]\!]_{R,s,h} \quad \text{for corresponding } df_i \text{ and } it_i. \end{array}$$

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.

For binary relations $t \sim t'$ between integers, sets, and multisets, including equality, the pure property plays an important role. Remember that in *SL* all terms are pure. To be consistent with *SL*, if both t and t' are pure, it is interpreted in the normal way. Otherwise, $t \sim t'$ is only defined on the minimum heaplet required by t and t', more concretely the union of the heaplet associated with t and t'.

$$\begin{array}{l} (R, s, h) \models t \sim t' \quad \text{iff} \quad t \text{ or } t' \text{ is pure and } [\![t]\!]_C \sim [\![t']\!]_C \\ \quad \text{or} \quad t \text{ and } t' \text{ are impure and there exist } R_1, R_2 \text{ s.t. } R = R_1 \cup R_2 \\ \quad \text{and } [\![t]\!]_{C|R_1} \neq \mathtt{undef}, [\![t']\!]_{C|R_2} \neq \mathtt{undef} \text{ and } [\![t]\!]_{C|R_1} \sim [\![t']\!]_{C|R_2} \end{array}$$

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 φ_0 and φ_1 hold respectively:

$$\begin{array}{l} (R, s, h) \models \varphi_0 * \varphi_1 \quad \text{iff} \quad \text{there exist } R_0, R_1 \text{ s.t. } R_0 \cap R_1 = \emptyset \text{ and} \\ R_0 \cup R_1 = R \text{ and } (R_0, s, h \mid R_0) \models \varphi_0 \text{ and } (R_1, s, h \mid R_1) \models \varphi_1 \end{array}$$

Boolean combinations are defined in the standard way:

$$\begin{array}{rcl} (R, s, h) \models \varphi_0 \wedge \varphi_1 & \text{iff} & (R, s, h) \models \varphi_0 \text{ and } (R, s, h) \models \varphi_1 \\ (R, s, h) \models \varphi_0 \vee \varphi_1 & \text{iff} & (R, s, h) \models \varphi_0 \text{ or } (R, s, h) \models \varphi_1 \\ (R, s, h) \models \neg\varphi & \text{iff} & (R, s, h) \not\models \varphi \end{array}$$

Semantics of recursive definitions

The main semantical difference between Dryad and *SL* is on recursive definitions. We would like to *deterministically delineate* the heap domain for any recursive definition, so that the heap domain required by any Dryad formula can be *syntactically determined*. Given a recursive definition $rec^{\Delta}_{\overrightarrow{pf},\vec{v}}$, the subscripts $\overrightarrow{pf}$ and $\vec{v}$ play a role in delineating the heap domain. Intuitively, the heap domain for $rec^{\Delta}_{\overrightarrow{pf},\vec{v}}(l)$ is the set of locations reachable from l using pointer-fields in $\overrightarrow{pf}$, but *without* going through the locations $\vec{v}$. In other words, we want to take the set of locations that lie *in between* l and $\vec{v}$. Precisely, this set is determined by a location l and a program state (R, s, h).

We denote it as $\text{reachset}_{\vec{pf},\vec{v}}(l,(R,s,h))$. Formally it is the smallest set of locations L satisfying the following two conditions:

1. l is in the set L if l is not in $\vec{v}$ and $l \neq nil$;
2. for each c in L, with $c \in R$, and for each pointer pf, if $h(c, pf)$ is not in $\vec{v}$ and is not nil, then $h(c, pf)$ is also in L.

Note that even though the reach set is defined with respect to the edges in the heaplet, we can determine whether R includes all nodes reachable from l without going through $\vec{v}$ in the *global heap* by checking whether $R = \text{reachset}_{\vec{pf},\vec{v}}(l,(R,s,h))$.

For each recursive definition $rec^{\Delta}_{\vec{pf},\vec{v}}$, we usually simply denote $\text{reachset}_{\vec{pf},\vec{v}}$ as reachset^{rec}, as the subscripts are implicitly known.

Now, given a program state $C = (R,s,h)$ and a recursive function/predicate rec^{Δ}, the semantics on a location l depends on whether the heap domain R is exactly the required reach set $\text{reachset}^{rec}(l,(R,s,h))$. If this is not true, we simply interpret it as `undef` or `false`.

If the heap domain matches the reach set (i.e., $R = \text{reachset}_{\vec{pf},\vec{v}}(l, (R, s, h))$), the semantics is defined in the natural way (using least fixed-points). The colon operator in the syntax of recursive function $f^{\Delta}_{\vec{pf},\vec{v}}$ translates into a nested if-then-else (ITE) operator. Formally,

$$[\![f^{\Delta}_{\vec{pf},\vec{v}}(l)]\!]_C = \text{ITE}(\varphi^f_1, [\![t^f_1]\!]_{C|R_1},\ \text{ITE}(\varphi^f_2, [\![t^f_2]\!]_{C|R_2}, \dots [\![t^f_{k+1}]\!]_{C|R_{k+1}} \dots))$$

where $R_1 \dots R_{k+1} \subseteq R$ such that $[\![t^f_i]\!]_{C|R_i} \neq \texttt{undef}$. In order to give least fixed-point semantics for recursive definitions in the logic, we extend the primitive data-types to lattice domains. *Bool* with the order $\texttt{false} \sqsubseteq \texttt{true}$ forms a complete lattice, and $\mathcal{S}(Loc)$ and $\mathcal{S}(Int)$ ordered by inclusion, with join as union and meet as intersection, form complete lattices. Integers and multisets are extended to lattices. Let $(Int_L, \leq)$ denote the complete lattice, where $Int_L = Int \cup \{-\infty, \infty\}$, and where the ordering is $\leq$, join is *max*, meet is *min*. Also, $\mathcal{MS}(Int)_L, \sqsubseteq$ denote the complete lattice constructed from $\mathcal{MS}(Int)$, where $\mathcal{MS}(Int)_L = \mathcal{MS}(Int) \cup \{\top\}$, and $\sqsubseteq$ extends the inclusion relation with $S \sqsubseteq \top$ for any $M \in \mathcal{MS}(Int)$. It is easy to see that $(Int_L, \sqsubseteq)$ and $(\mathcal{MS}(Int)_L, \sqsubseteq)$ are complete lattices.

Formally, let *Def* consists of definitions of integer functions *I*, set-of-locations functions *SL*, set-of-integers functions *SI*, multiset-of-integers functions *MSI* and predicates *P*. Since these definitions could rely on each other, we evaluate them altogether as a function vector

$$r^{\Delta} = (\overrightarrow{i^{\Delta}}, \overrightarrow{sl^{\Delta}}, \overrightarrow{si^{\Delta}}, \overrightarrow{msi^{\Delta}}, \overrightarrow{p^{\Delta}})$$

We take the cartesian product lattice of the individual lattices and take the least fixed-point of r^{Δ} to obtain the semantics for each definition. Let $\text{select}_{rec}(lfp(r^{\Delta}))$, for each recursive definition rec^{Δ}, denote the selection of the coordinate for rec^{Δ} in $lfp(r^{\Delta})$.

Now we can formally define the semantics of recursive definitions. For any configuration C, the semantics of a recursive function f^{Δ} is defined as:

$$[\![f^{\Delta}(lt)]\!]_C = \begin{cases} \text{select}_f(lfp(r^{\Delta}))([\![lt]\!]_C) & \text{if } R = \text{reachset}^f([\![lt]\!]_C, C) \\ \texttt{undef} & \text{otherwise} \end{cases}$$

and the semantics of a recursive predicate p^{Δ} is defined as

$$[\![p^{\Delta}(lt)]\!]_C = \begin{cases} \text{select}_p(lfp(r^{\Delta}))([\![lt]\!]_C) & \text{if } R = \text{reachset}^p([\![lt]\!]_C, C) \\ \texttt{false} & \text{otherwise} \end{cases}$$

Remark: Note that we disallow negative operations (subtraction, set-difference and negation) in defining recursive definitions. This syntactical restriction guarantees that each iteration of r^{Δ} is monotonic. By Knaster-Tarski theorem, r^{Δ} admits a least fixed-point.

Examples

The Dryad logic was already used in Section 3 to define max-heaps. Note that the definition of a max-heap is 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.

To clarify the difference between Dryad and *SL*, consider now this recursive definition:

$$p^{\Delta}_{\{l,r\}}(x) \stackrel{def}{=} (x = \texttt{nil} \wedge \texttt{emp}) \ \vee\ \Big[(x \xrightarrow{l,r} y, z) \ *\ \big(p^{\Delta}_{\{l,r\}}(y) \vee p^{\Delta}_{\{l,r\}}(z)\big)\Big]$$

Now consider a global heap that has a tree rooted at x with pointer fields l and r. The above recursive formula, in separation logic, will be true on any heaplet that contains the nodes of a path in this tree from x to *nil*. However, in Dryad, we require that the heaplet must satisfy the heap constraints of the formula and also be the precise set of locations reachable from x using the pointer fields l and r. Consequently, if the tree pointed to by x has more than one path, the Dryad formula will be *false for any heaplet*.

The above example shows the advantage of Dryad; when heaplets are determined, we can avoid quantification. We have not found natural examples where an undetermined heaplet semantics helps in specifying properties of heaps.

Dryad 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, Dryad can express structures like cyclic lists and doubly-linked lists.

A cyclic-list is captured as $(v \mapsto y) * lseg^{\Delta}_{next,v}(y)$. Here, v is a program variable which denotes the head of the cyclic-list and $lseg^{\Delta}_{next,v}(y)$ captures the list segment from y back to the head v, where the subscripts *next* and v indicate that the heaplet of the list segment is the locations that can be reached using the field *next*, but without going through v:

$$lseg^{\Delta}_{next,v}(y) \stackrel{def}{=} (y = v \wedge \texttt{emp}) \vee (\,(y \stackrel{next}{\mapsto} z) * lseg^{\Delta}_{next,v}(z)\,)$$

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

$$\begin{aligned} dll^{\Delta}_{next}(x) \quad = \quad & (x = \texttt{nil} \wedge \texttt{emp}) \ \vee\ (x \stackrel{next}{\mapsto} \texttt{nil}) \ \vee \\ & (\, x \stackrel{next}{\mapsto} y \ *\ ((y \stackrel{prev}{\mapsto} x * \texttt{true}) \ \wedge\ dll^{\Delta}_{next}(y))\,) \end{aligned}$$

The first two disjuncts in the definition cover the base case when x is *nil* or the location next to x is *nil*; otherwise, let y be the location next to x, 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 Dryad logic 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 treeness, using 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 $mheap^{\Delta}(x) * mheap^{\Delta}(y)$, where $mheap^{\Delta}$ is defined in Section 3. Since the heaplets for $mheap^{\Delta}(x)$ and $mheap^{\Delta}(y)$ are precise, it can get translated to an equivalent formula with a *free* set variable G that denotes the global heap over which the formula is evaluated:

$$\begin{aligned} & mheap(x) \wedge mheap(y) \wedge (reach^{mheap}(x) \cap reach^{mheap}(y) = \emptyset) \\ & \qquad \wedge (reach^{mheap}(x) \cup reach^{mheap}(y) = G) \end{aligned}$$

where *mheap* and $reach^{mheap}$ are corresponding recursive definitions in classical logic, which will be defined later in this section. Note that we use italics and remove the $^{\Delta}$ superscript to show the difference from their counterpart in Dryad.

We assume the Dryad formula to be translated is in *disjunctive normal form*, i.e., $\vee$ operators should be above all $*$ and $\wedge$ operators.

Construct	Domain-exact	Scope
$var/const$	$\texttt{false}$	$\emptyset$
$\{t\}/\{t\}_m$	dom-ext(t)	scope(t)
t op t'	dom-ext(t) $\vee$ dom-ext(t')	scope(t) $\cup$ scope(t')
$f^{\Delta}(lt)$	$\texttt{true}$	reachset$^f(lt)$
$\texttt{true/false}$	$\texttt{false}$	$\emptyset$
$\texttt{emp}$	$\texttt{true}$	$\emptyset$
$lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it})$	$\texttt{true}$	$\{lt\}$
$p^{\Delta}(lt)$	$\texttt{true}$	reachset$^p(lt)$
$t \sim t'$	dom-ext(t) $\vee$ dom-ext(t')	scope(t) $\cup$ scope(t')
$\varphi \wedge \varphi'$	dom-ext(φ) $\vee$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')
$\varphi * \varphi'$	dom-ext(φ) $\wedge$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')

Figure 3. Domain-exact property and Scope function. Both are defined only for terms and formulas without disjunction and negation. A formula is assumed in its disjunctive normal form.

This is not a real 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 separating conjunction $P*Q$ is translated to the following set constraint: the intersection of the sets associated with the heaplets in the formulas P and Q is empty. Given a formula φ in DRYAD and its associated heap domain modeled by a set variable G, we define an inductive translation T into a classical logic formula $T(\varphi,\ G)$ in the quantifier-free theory of finite sets, integers and uninterpreted functions. The translated formula is not interpreted on a heaplet, but interpreted on a global heap (i.e., with the heap domain Loc).

The translation uses an auxiliary *domain-exact* property and an auxiliary *scope* function. The domain-exact property indicates whether a term evaluates to a well-defined value or a positive formula evaluates to true on a fixed heap domain or not. This is different from the property *pure*; a pure formula or term is not domain-exact but the reverse implication is not true, in general. For example, the formula $(lt \longmapsto it) * true$ is not domain-exact but is also not pure. The scope function maps a term to the minimum heap domain required to evaluate it to a normal value, and maps a positive formula to the minimum heap domain required to evaluate it to *true*. The domain-exact property and the scope function are defined inductively in Figure 3.

We describe the logic translation in detail in Figure 4. The ITE expression used in the translation is short for "if-then-else". It is just a conditional expression defined as follows: ITE(ϕ, t_1, t_2) evaluates to t_1 if ϕ is true, otherwise evaluates to t_2.

In general, our translation restricts an impure term/formula to be evaluated only on the syntactically determined heap domain according to the semantics of DRYAD. In particular, when evaluating a recursive formula or predicate p^{Δ}, we ensure that the heaplet is precisely the reach set $reach^p(lt)$. For a formula $\varphi * \varphi'$, translation to classical logic depends on whether the sub-formulas φ and φ' are domain-exact or not. If a sub-formula is domain-exact then it is evaluated on its scope. If it is not domain-exact, then it is evaluated on the rest of the heaplet.

Recursive definitions in DRYAD are also translated to recursive definitions in classical logic. Translating a recursive definition rec^{Δ} uses the corresponding definitions rec and $reach^{rec}$, both of which are defined recursively in classical logic. The set $reach^{rec}$ represents the domain of the required heaplet for evaluating rec^{Δ}, and the Δ-eliminated definition rec captures the value of rec^{Δ} when the

$$
\begin{array}{rcl}
T(var\ /\ const,\ G) & \equiv & var\ /\ const \\
T(\{t\}\ /\ \{t\}_m,\ G) & \equiv & \{t\}\ /\ \{t\}_m \\
T(t\ \text{op}\ t',\ G) & \equiv & T(t, G)\ \text{op}\ T(t', G) \\
T(f^{\Delta}(lt),\ G) & \equiv & \text{ITE}\ (reach^f(lt) = G,\ f(lt),\ \texttt{undef}) \\
T(\texttt{true}\ /\ \texttt{false},\ G) & \equiv & \texttt{true}\ /\ \texttt{false} \\
T(\texttt{emp},\ G) & \equiv & G = \emptyset \\
T(lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it}),\ G) & \equiv & G = \{lt\} \wedge \bigwedge_{pf_i}\ pf_i(T(lt, G)) = T(lt_i, G) \\
 & & \qquad \wedge \bigwedge_{df_i}\ df_i(T(lt, G)) = T(it_i, G) \\
T(p^{\Delta}(lt),\ G) & \equiv & p(lt) \wedge G = reach^p(lt) \\
T(t \sim t',\ G) & \equiv & \begin{cases} t \sim t' & \text{if } t \sim t' \text{ is not domain-exact} \\ t \sim t' \wedge G = \text{scope}(t \sim t') & \text{otherwise} \end{cases} \\
T(\varphi \wedge \varphi',\ G) & \equiv & T(\varphi, G) \wedge T(\varphi', G) \\
T(\varphi \vee \varphi',\ G) & \equiv & T(\varphi, G) \vee T(\varphi', G) \\
T(\neg\varphi,\ G) & \equiv & \neg T(\varphi, G)
\end{array}
$$

$$
T(\varphi * \varphi',\ G) \quad \equiv \quad
\begin{cases}
T(\varphi,\ \text{scope}(\varphi))\ \wedge\ T(\varphi',\ \text{scope}(\varphi')) \\
\quad \wedge\ \text{scope}(\varphi) \cup \text{scope}(\varphi') = G \\
\quad \wedge\ \text{scope}(\varphi) \cap \text{scope}(\varphi') = \emptyset \\
\qquad\qquad \text{if both } \varphi \text{ and } \varphi' \text{ are domain-exact} \\[4pt]
T(\varphi,\ \text{scope}(\varphi))\ \wedge\ T(\varphi',\ G \setminus \text{scope}(\varphi)) \\
\quad \wedge\ \text{scope}(\varphi) \subseteq G \qquad \text{if only } \varphi \text{ is domain-exact} \\[4pt]
T(\varphi',\ \text{scope}(\varphi'))\ \wedge\ T(\varphi,\ G \setminus \text{scope}(\varphi')) \\
\quad \wedge\ \text{scope}(\varphi') \subseteq G \qquad \text{if only } \varphi' \text{ is domain-exact} \\[4pt]
T(\varphi,\ \text{scope}(\varphi))\ \wedge\ T(\varphi',\ \text{scope}(\varphi')) \\
\quad \wedge\ \text{scope}(\varphi) \cup \text{scope}(\varphi') \subseteq G \\
\quad \wedge\ \text{scope}(\varphi) \cap \text{scope}(\varphi') = \emptyset \\
\qquad\qquad \text{if neither } \varphi \text{ nor } \varphi' \text{ is domain-exact}
\end{cases}
$$

Figure 4. Translating DRYAD terms/formulas to classical logic

heaplet is restricted to $reach^{rec}$. Formally, suppose rec^{Δ} is a recursive definition w.r.t. pointer fields $\vec{pf}$ and stopping locations $\vec{v}$, then $reach^{rec}$ is recursively defined as the least fixed-point of

$$reach^{rec}(x) \overset{def}{=} \text{ITE}\Big(x = \texttt{nil} \vee x \in \vec{v},\ \emptyset,\ \{x\} \cup \bigcup_{pf \in \vec{pf}} (reach^{rec}(pf(x))) \Big)$$

For each recursive predicate p^{Δ} defined as $p^{\Delta}(x) \overset{def}{=} \varphi^p(x, \vec{v}, \vec{s})$, we define

$$p(x) \overset{def}{=} T(\ \varphi^p(x,\ \vec{v}, \vec{s}),\ reach^p(x)\ )$$

Similarly, for each recursive function f^{Δ} defined as

$$f^{\Delta}(x) \overset{def}{=} (\varphi_1^f(x, \vec{v}, \vec{s}) : t_1^f(x, \vec{s})\ ;\ \ldots\ \varphi_k^f(x, \vec{v}, \vec{s}) : t_k^f(x, \vec{s})\ ;\ \text{default} : t_{k+1}^f(x, \vec{s}))$$

we define

$$
\begin{array}{rl}
f(x) \overset{def}{=} & \text{ITE}\Big(T(\varphi_1^f(x, \vec{v}, \vec{s}), reach^f(x)),\ \ t_1^{f-\Delta}(x, \vec{s}) \\
& \qquad \text{ITE}\Big(T(\varphi_2^f(x, \vec{v}, \vec{s}), reach^f(x)),\ \ t_2^{f-\Delta}(x, \vec{s}) \\
& \qquad\qquad \ldots, t_{k+1}^{f-\Delta}(x, \vec{s})\ \ \ldots \Big)\Big)
\end{array}
$$

where $t_i^{f-\Delta}(x, \vec{s})$ is just the classical logic counterpart of $t_i^f(x, \vec{s})$, when interpreted in a heap domain within $reach^f(x)$. Formally it is short for

$$\text{ITE}\Big(\text{scope}(t_i^f(x, \vec{s})) \subseteq reach^f(x),\ T(t_i^f(x, \vec{s}),\ \text{scope}(t_i^f(x, \vec{s}))),\ \texttt{undef}\Big)$$

Now for each set of recursive definitions Def^{Δ} in DRYAD, we can translate it to a set of recursive definitions Def in classical logic.

Theorem 5.1. *Let φ be a* DRYAD *formula w.r.t. a set of recursive definitions Def^{Δ}. For every program state C with heap domain Loc, and for every interpretation of variables $\mathcal{I}$ including a valuation for set-variable G, $(C,\ \mathcal{I}) \models T(\varphi,\ G)$ w.r.t. Def if and only if $(C\mid_G,\ \mathcal{I} \setminus \{G\}) \models \varphi$ w.r.t. Def^{Δ}.* $\square$

$$
\begin{array}{lll}
P & :- & P\,;P \mid stmt \\
stmt & :- & u := v \mid u := \texttt{nil} \mid u := v.pf \mid u.pf := v \\
 & & \mid\ j := u.df \mid u.df := j \mid j := aexpr \\
 & & \mid\ u := \texttt{new} \mid \texttt{free}\ u \mid \texttt{assume}\ bexpr \\
 & & \mid\ u := f(\vec{v}, \vec{z}) \mid j := g(\vec{v}, \vec{z}) \\
aexpr & :- & int \mid j \mid aexpr + aexpr \mid aexpr - aexpr \\
bexpr & :- & u = v \mid u = \texttt{nil} \mid aexpr \le aexpr \\
 & & \mid\ \neg bexpr \mid bexpr \vee bexpr
\end{array}
$$

Figure 5. Syntax of programs

6. Natural Proofs for Dryad

In this section we show how Dryad 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 Dryad. We first introduce a simple programming language and the corresponding Hoare-triples, generate a classical logic formula as the verification condition, utilizing in part the translation defined in Section 5. Then we present the natural proof framework which consists of two steps. In the first step, we utilize the idea of unfolding across the footprint to strengthen the verification condition. In the second step, 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 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 5, 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 $f(\vec{v}, \vec{z})$ and one returning an integer $g(\vec{v}, \vec{z})$. Each procedure/program is annotated with its pre- and post-conditions in Dryad. The pre-condition is denoted as a formula $\psi_{pre}(\vec{v}, \vec{z}, \vec{c})$, where $\vec{v}$ and $\vec{z}$ are variables as the formal parameters/program variables, $\vec{c}$ is a set of implicitly existentially quantified complimentary variables (e.g., variable K in the pre-condition φ_{pre} in Figure 1). The post-condition is denoted as a formula $\psi_{post}(ret, \vec{v}, \vec{z}, \vec{c})$, where *ret* is the variable representing the returned value, of corresponding type, $\vec{v}$ and $\vec{z}$ are program variables, $\vec{c}$ is a set of complimentary variables that have appeared in the pre-condition ψ_{pre}.

Given a straight-line program with its pre- and post-conditions $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, we define its partial correctness without considering memory errors[3]: P is partially correct iff for every normal execution (memory-error free) of P, which transits state C to state C', if $C \models \psi_{pre}$, then $C' \models \psi_{post}$.

Given a Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$ as defined above, a set of recursive definitions and a set of annotated procedure declarations are presented here. Assume that P consists of n statements, then consider a normal execution $\mathcal{E}$, which can be represented as a sequence of program states $(C_0, \ldots, C_n)$, where each $C_i = (R_i, s_i, h_i)$ represents the program state after executing the first i statements. The verification condition is just a formula interpreted on a state sequence $(C_0, \ldots, C_n)$. Let $pf_i : Loc \to Loc$ be the function mapping every location l to its pf pointer, i.e., $pf_i(l) = h_i(l, pf)$ for every location l. Similarly, $df_i : Loc \to Int$ is defined such that $df_i(l) = h_i(l, df)$ for every l. Recall that every program variable is either pre-assigned or assigned once in the program, each s_i is an expantion of the previous one, and s_n is the store with all program variables defined. Hence we simply use v to denote $s_n(v)$. 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 $C_i \models T(p^{\Delta}(l), \mathrm{reachset}^{p}(l))$. Now for every formula φ 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 $\varphi[i]$.

We algorithmically derive the verification condition ψ_{VC} corresponding to it in classical logic with recursive definitions on the global heap (the algorithm is quite involved, and is presented in Appendix A in the supplemental material).

Theorem 6.1. *Given a Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, assume that each procedure call in P satisfies its associated pre- and post-conditions. Then the triple is valid if the formula ψ_{VC} derived above is valid. Moreover, when P contains no procedure calls, the triple is valid iff ψ_{VC} is valid.*

Proof. Presented in Appendix B in the supplemental material. □

6.2 Unfolding Across the Footprint

The verification condition obtained above is a quantifier-free formula involving recursive definitions and the reachable sets of the form $reach^{p}(x)$, which are also defined recursively. While these recursive definitions can be unfolded *ad infinitum*, we exploit a proof tactic called *unfolding across the footprint*. 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. The idea is to unfold the recursive definitions over the footprint of the program, so that recursive definitions on the footprint nodes are related, as precisely 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 unfolding formulas ensure tight conditions that the frontier nodes have to satisfy.

Furthermore, to enable effective frame reasoning, it is also necessary to strengthen the verification condition with a set of instances of the frame rule. More concretely, we need to capture the fact that a recursive definition (or a field) on a location is unchanged during a segment or procedure call of the program, if the reachable locations (or only the location itself) are not affected by the segment or procedure call.

We incorporate the above facts formally into the verification condition. Let us introduce a macro function fp that identifies the location variables that are in (or aliased to something in) the footprint. The footprint of P, FP, is the set of dereferenced variables in P (we call a location variable dereferenced if it appears on the left-hand side of a dereferencing operator "." in P). Then $fp(u) \equiv \bigvee_{v \in FP}(u = v)$.

Now we state the unfoldings and framings using a formula UnfoldAndFrame. 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. Then

$$
\begin{aligned}
\textsc{UnfoldAndFrame} \equiv & \bigwedge_{rec} \bigwedge_{0 \le d \le m} \bigwedge_{u \in LVars \cup \{nil\}} \Big[\\
& \Big(\big(fp(u) \vee u = nil \big) \Rightarrow \big(\textsc{Unfold}_{d}^{rec}(u) \wedge \textsc{FieldUnchanged}_{d}(u) \big) \Big) \wedge \\
& \Big(\big(\neg fp(u) \vee u = nil \big) \Rightarrow \textsc{RecUnchanged}_{d}^{rec}(u) \Big) \Big]
\end{aligned}
$$

The formula enumerates every recursive definition *rec* and every index d, and for each location u that is either pointed to by a location variable or is *nil*, the formula checks if u is in the footprint, and then unfolds it or frames it accordingly. If u is in the footprint, then we unfold *rec* for the timestamps before and after S_d (represented by the formula $\textsc{Unfold}_{d}^{rec}(u)$); moreover, all fields of

[3] 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.

u are unchanged if it is not affected during calling g_d (represented by the formula $\textsc{FieldUnchanged}_d(u)$). If u is not in the footprint, i.e., in the frontier, then *rec* and its corresponding reach set $reach^{rec}$ are unchanged after executing S_d, if S_d does not modify any location in $reach^{rec}$; they are also unchanged if $reach^{rec}$ is not affected by calling g_d. These frame assertions are represented by the formula $\textsc{RecUnchanged}_d^{rec}(u)$. All the sub-formulas mentioned above are formulated in Appendix C in the supplemental material.

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

$$\psi'_{\mathrm{VC}} \equiv \psi_{\mathrm{VC}} \wedge \textsc{UnfoldAndFrame}$$

Since the incorporated formula is implied by the verification condition, we can reduce the validity of ψ_{VC} to the validity of ψ'_{VC}.

Theorem 6.2. *Given a Hoare-triple $\{\psi_{pre}\}$ P $\{\psi_{post}\}$, its verification condition ψ_{VC} is valid if and only if ψ'_{VC} is valid.* □

6.3 Formula Abstraction

While checking the validity of the strengthened verification condition ψ'_{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]. This tactic roughly corresponds to applying *unification* in proof systems.

To prove ψ'_{VC}, we first replace each recursive predicate rec_d with an uninterpreted predicate $\hat{rec}_d$, and replacing the corresponding reach-set function $reach_d^{rec}$ with an uninterpreted function $\hat{reach}_d^{rec}$. Let the result formula be $\psi_{\mathrm{VC}}^{\mathrm{abs}}$. This conversion, called *formula abstraction*, is sound: if $\psi_{\mathrm{VC}}^{\mathrm{abs}}$ is valid, so is ψ'_{VC}. When a proof for $\psi_{\mathrm{VC}}^{\mathrm{abs}}$ is found, we call it a natural proof for ψ'_{VC} (and also for ψ_{VC}).

The formula abstraction step is the only step that introduces incompleteness in our framework, but helps us transform the verification condition to a decidable theory. Formula abstraction (combined with unfolding recursive definitions across the footprint) discovers *recursive proofs* where the recursion is structural recursion on the definitions of the data-structures. The use of these tactics comes from the observation that such programs often have such recursive proofs (see [39] also for use of formula abstractions).

Our goal now is to check the satisfiability of $\neg\psi_{\mathrm{VC}}^{\mathrm{abs}}$ in a decidable theory. The resulting formula can be expressed using the theory of maps (to model sets) and corresponding map operations to model set operations. Formulas of the kind $S_1 \leq S_2$, where S_1 and S_2 are sets/multi-sets of integers, are the only ones that introduce quantification, but they can be translated to formulas in the array property fragment, which is decidable [12]. We hence obtain a formula ψ^{APF} in the array property fragment combined with the theory of uninterpreted functions, maps, and arithmetic (details are in Appendix D in the supplementary material).

Theorem 6.3. *Given a Hoare-triple $\{\psi_{pre}\}$ P $\{\psi_{post}\}$, if the derived array formula ψ^{APF} is satisfiable, then the Hoare-triple is valid.* □

User-provided axioms:

While natural proofs are often effective in finding recursive proofs that unfold recursive definitions and do unification, they are not geared towards finding *relationships* between various recursive definitions themselves. We hence require the user to provide certain obvious relationships between the different recursive definitions as axioms. For example, $lseg(x, y) * list(y) \Rightarrow list(x)$ is such an axiom saying that a list segment concatenated with a list yields a list. Note that these axioms are *not* program-dependent, and hence are not program-specific tactics that the user provides. These axioms are necessary typically to relate partial data-structure properties (like list segments) to complete ones (like lists), especially in iterative programs (as opposed to recursive ones), and we can fix them for each class of data-structures. We also allow the use of the separating implication, $-*$, from separation logic while specifying these axioms. User-defined axioms are instantiated, using the natural proof philosophy, on precisely the footprint nodes uniformly, and get translated to quantifier-free formulas.

Data-structure	Routine	Time (s) / Routine
Singly-Linked List	`find_rec, insert_front, insert_back_rec, delete_all_rec, copy_rec, append_rec, reverse_iter`	< 1s
Sorted List	`find_rec, insert_rec, merge_rec, delete_all_rec, insert_sort_rec, reverse_iter, find_last_iter`	< 1s
	`insert_iter`	1.4
	`quick_sort_iter`	64.8
Doubly-Linked List	`insert_front, insert_back_rec, delete_all_rec, append_rec, mid_insert, mid_delete, meld`	< 1s
Cyclic List	`insert_front, insert_back_rec, delete_front, delete_back_rec`	< 1s
Max-Heap	`heapify_rec`	8.8
BST	`find_rec, find_iter, insert_rec, delete_rec, remove_root_rec`	< 1s
	`insert_iter`	72.4
	`find_leftmost_iter`	4.7
	`remove_root_iter`	65.6
	`delete_iter`	225.2
Treap	`find_rec, delete_rec`	< 1s
	`insert_rec`	12.7
	`remove_root_rec`	9.5
AVL-Tree	`balance, leftmost_rec`	< 1s
	`insert_rec`	4.1
	`delete_rec`	13.9
RB-Tree	`insert_rec`	73.9
	`insert_left_fix_rec`	8.1
	`insert_right_fix_rec`	5.1
	`delete_rec`	12.1
	`delete_left_fix_rec`	7.6
	`delete_right_fix_rec`	5.5
	`leftmost_rec`	< 1s
Binomial Heap	`find_min_rec`	1.1
	`merge_rec`	152.7
Schorr-Waite (for trees)	`marking_iter`	< 1s
Tree Traversals	`inorder_tree_to_list_rec`	2.4
	`inorder_tree_to_list_iter`	42.7
	`preorder_rec, postorder_rec`	< 1s
	`inorder_rec`	3.76

Figure 6. Results of verifying data-structure algorithms. (more details at `http://www.cs.uiuc.edu/~madhu/dryad/sl/`)

7. Experimental Evaluation

We have implemented a prototype of the natural proof methodology for Dryad presented in this paper. The prototype 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 (or *basic blocks*) 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. For each basic block, the verifier automatically generates the abstracted formula ψ^{APF} as described in Section 6, and passes ψ^{APF} to Z3 [17], a state-of-the-art SMT solver, to check the satisfiability in the decidable theory of array property fragment. The front-end of our verifier is based on

ANTLR and our tool is around 4000 lines of C# code. Using the verifier, we successfully proved the partial correctness of 59 routines over a large class of programs involving heap data structures like sorted lists, doubly-linked lists, cyclic lists and trees. Additionally, we pit our natural proofs methodology against real-world programs and successfully verified, in total, 47 routines from different projects including the list and queue implementations in the Glib open source library, the OpenBSD library, the Linux kernel and the memory regions and the page cache implementations from two different operating systems. Experimental details are available at `http://www.cs.uiuc.edu/~madhu/dryad/sl/` .

We conducted the experiments on a machine with a dual-core, 2.4GHz CPU and 6GB RAM. The first part of our experimental results is tabulated in Figure 6. In general, for every routine, we checked the properties formalizing the complete verification of the routines— capturing the precise structure of the resulting heap-structure, the precise change to the data stored in the nodes and the precise heaplet modified by the respective routines.

For every routine, the suffix `rec` or `iter` indicates if the routine was implemented recursively or iteratively using while loops. The names for most of the routines are self-descriptive. Routines like `find`, `insert`, `delete`, `append`, etc. are the natural implementations of the corresponding data structure operations. The routine `delete_all` for singly-linked lists, sorted lists and doubly-linked lists recursively deletes all occurrences of a particular key in the input list. 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. The routine `remove_root` for binary search trees and treaps is an auxiliary routine which is called in `delete`. Similarly, the routines `leftmost` for AVL-trees and RB-trees and `delete_fix` and `insert_fix` for RB-trees are also auxiliary routines.

Schorr-Waite is a well-known graph marking algorithm which marks all the reachable nodes of the graph using very little additional space. The algorithm achieves this by manipulating the pointers in the graph such that the stack of nodes along the path from the root is encoded in the graph itself. The Schorr-Waite algorithm is used in garbage collectors and it is traditionally considered as a challenging problem for verification [19]. The routine `marking` is an implementation of Schorr-Waite for trees [24] and we check the property that the resulting output tree is well-marked.

The routines `inorder_tree_to_list` construct a list consisting of the keys of the input tree, which is traversed inorder. The iterative version of this algorithm achieves this by maintaining a worklist/stack of sub-trees which remain to be processed at any given time. The routines `inorder`, `preorder` and `postorder` number the nodes of an input tree according to the inorder, preorder and postorder traversal algorithm, respectively.

Figure 7 shows the results of applying natural proofs to the verification of various other real world programs and libraries. Glib is the low-level C library that forms the basis of the GTK+ toolkit and the GNOME desktop environment, apart from other open source projects. Using our prototype verifier, we efficiently verified Glib implementation of various routines for manipulating singly-linked and doubly-linked lists. We also verified the queue library which forms part of the OpenBSD operating system.

ExpressOS is an operating-system/browser implementation which provides security guarantees to the user via formal verification [29]. The module `cachePage` maintains a cache of the recently used disc pages. The cache is implemented as a priority queue based on a sorted list. We prove that the methods `add_cachepage` and `lookup_prev` (both called whenever a disc page is accessed) maintain the sortedness property of the cache page.

Example	Routine	Time (s) / Routine
glib/gslist.c Singly Linked-List LOC: 1.1K	`free, prepend, concat, insert_before, remove_all, remove_link, delete_link, copy, reverse, nth, nth_data, find, position, index, last, length`	< 1s
	`append`	4.9
	`insert_at_pos`	11.4
	`remove`	3.1
	`insert_sorted_list`	16.6
	`merge_sorted_lists`	6.1
	`merge_sort`	3.0
glib/glist.c Doubly Linked-List LOC: 0.3K	`free, prepend, reverse, nth, nth_data, position, find, index, last, length`	< 1s
OpenBSD/queue.h Queue LOC: 0.1K	`simpleq_init, simpleq_remove_after`	< 1s
	`simpleq_insert_head`	1.6
	`simpleq_insert_tail`	3.6
	`simpleq_insert_after`	18.3
	`simpleq_remove_head`	2.1
ExpressOS/cachePage.c LOC: 0.1K	`lookup_prev`	2.4
	`add_cachepage`	6.4
ExpressOS/ memoryRegion.c LOC: 0.1K	`memory_region_init`	< 1s
	`create_user_space_region`	3.6
	`split_memory_region`	5.8
linux/mmap.c LOC: 0.1K	`find_vma, remove_vma, remove_vma_list`	< 1s
	`insert_vm_struct`	11.6

Figure 7. Results of verifying open-source libraries. (more details at `http://www.cs.uiuc.edu/~madhu/dryad/sl/`)

In an OS kernel, a process address space consists of a set of intervals of linear addresses represented as a *memory region*. In the ExpressOS implementation, a memory region is implemented as a sorted doubly-linked list where each node of the list with a *start* and an *end* address represents an interval included in the address space. We also verified some key components of the Linux implementation of a memory region, present in the file *mmap.c*. In Linux, a memory region is represented as a red-black tree where each node, again, represents an address interval. We proved methods which find, remove and insert a *vma_struct* (*vma* is short for virtual memory address) into a memory region.

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 $*$ in the specification between two disjoint heaplets, 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.

Our experiments show that the natural proof methodology set forth in this paper is successful in efficiently proving full-functional correctness of a large variety of algorithms. Most of the VCs generated for the above examples were discharged by Z3 in a few seconds. To the best of our knowledge, this is the first automatic mechanism that can prove such a wide variety of algorithms correct, handling such complex properties of structure, data and separation.

Acknowledgments

This work is partially funded by NSF CAREER award #0747041, NSF CCF #1018182, and NSA contract H98230-10-C-0294.

References

[1] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis of single-parent heaps. In *VMCAI'07*, volume 4349 of *LNCS*, pages 91–105. Springer, 2007.

[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] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, volume 4111 of *LNCS*, pages 115–137. Springer, 2005.

[6] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[7] N. Bjørner and J. Hendrix. Linear functional fixed-points. In *CAV'09*, volume 5643 of *LNCS*, pages 124–139. Springer, 2009.

[8] M. Botinčan, M. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. *ENTCS*, 254:5 – 23, 2009.

[9] A. Bouajjani, C. Drăgoi, 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.

[10] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI'11*, pages 578–589. ACM, 2011.

[11] M. Bozga, R. Iosif, and S. Perarnau. Quantitative separation logic and programs with lists. In *IJCAR'08*, volume 5195 of *LNCS*, pages 34–49. Springer, 2008.

[12] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI'06*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[13] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL'08*, pages 247–260. ACM, 2008.

[14] 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*, 77(9):1006 – 1036, 2012.

[15] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245. ACM, 2011.

[16] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[17] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[18] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[19] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *SEFM'05*, pages 190–199. IEEE-CS, 2005.

[20] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM'11*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

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

[22] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL'08*, pages 171–182. ACM, 2008.

[23] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[24] A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS'06*, volume 4134 of *LNCS*, pages 261–279. Springer, 2006.

[25] P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In *SAS'11*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.

[26] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *POPL'11*, pages 611–622. ACM, 2011.

[27] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL'12*, pages 123–136. ACM, 2012.

[28] 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.

[29] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *ASPLOS'13*, pages 293–304. ACM, 2013.

[30] G. Nelson. Verifying reachability invariants of linked structures. In *POPL'83*, pages 38–47. ACM, 1983.

[31] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE-CS, 2002.

[37] 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.

[38] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.

[39] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL'10*, pages 199–210. ACM, 2010.

[40] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS'11*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.

[41] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV'08*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

[42] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *PLDI'10*, pages 99–110. ACM, 2010.

[43] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI'08*, pages 349–361. ACM, 2008.

[44] K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *PLDI'09*, pages 338–351. ACM, 2009.