Natural Proofs for Data Structure Manipulation in C using Separation Logic

Edgar Pek
University of Illinois at Urbana-Champaign
pek1@illinois.edu

Xiaokang Qiu
MIT CSAIL
xkqiu@csail.mit.edu

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

Abstract

The natural proof technique for heap verification developed by Qiu et al. [32] provides a platform for powerful sound reasoning for specifications written in a dialect of separation logic called Dryad. Natural proofs are proof tactics that enable automated reasoning exploiting recursion, mimicking common patterns found in human proofs. However, these proofs are known to work only for a simple toy language [32].

In this work, we develop a framework called VCDRYAD that extends the VCC framework [9] to provide an automated deductive framework against separation logic specifications for C programs based on natural proofs. We develop several new techniques to build this framework, including (a) a novel tool architecture that allows encoding natural proofs at a higher level in order to use the existing VCC framework (including its intricate memory model, the underlying type-checker, and the SMT-based verification infrastructure), and (b) a synthesis of ghost-code annotations that captures natural proof tactics, in essence forcing VCC to find natural proofs using primarily decidable theories.

We evaluate our tool extensively, on more than 150 programs, ranging from code manipulating standard data structures, well-known open source library routines (Glib, OpenBSD), Linux kernel routines, customized OS data structures, etc. We show that all these C programs can be fully automatically verified using natural proofs (given pre/post conditions and loop invariants) without *any user-provided proof tactics*. VCDRYAD is perhaps the first deductive verification framework for heap-manipulating programs in a real language that can prove such a wide variety of programs automatically.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, Programming by contract; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Invariants, Mechanical verification, Pre- and post-conditions

Keywords program verifiers; data structures; natural proofs; separation logic

PLDI '14, June 09–11, 2014, Edinburgh, United Kingdom.
Copyright © 2014 ACM 978-1-4503-2784-8/14/06...$15.00.
http://dx.doi.org/10.1145/2594291.2594325

1. Introduction

A very promising mostly-automated yet scalable approach to program verification is the paradigm of *automated deductive verification*. Programmers develop code, say imperative code, in any language of their choice, and annotate the code with modular contracts, using pre-conditions, post-conditions, assertions, and invariants such as loop and object invariants. These annotations not only capture the specification of the software, but also provide invariants that chop up the reasoning of the program into Hoare triples involving finite loop-less code. Using a logical semantics of the programming language, program verification reduces to reasoning purely about logic validity. Finally, these validity checks can be automated, for the large part, using automated theorem provers and SMT solvers. The tools VCC [9], DAFNY [24], HAVOC [12], etc. that compile into BOOGIE [1] (which in turn generates verification conditions to the SMT solver Z3) and several other tools such as VeriFast [21], jStar [15], Smallfoot [4], etc. fall in this category. Several large software have been wholly or partly verified using such tools, including Microsoft Hypervisor [11], Verve [37] (an OS), and ExpressOS [26] (an Android platform verified for a small set of security properties).

One of the main drawbacks of state-of-the-art mostly-automated deductive verification tools today is that when verifying properties of the dynamically manipulated heap, the verification condition expressed in logic is typically not in a decidable theory, and hence the proof is hardly ever automatic. Tools that sit over BOOGIE (like VCC) and others such as VeriFast [21] and Bedrock [8] give the programmer the ability to write code-level proof tactics to make the proof go through. For example, VCC allows programmers to give *triggers* that essentially give terms that the underlying SMT solver should try substituting universally quantified variables with in order to prove the negation of a verification condition unsatisfiable. In VeriFast, the programmer often works with recursive definitions and can at the code-level ask for such definitions to be unfolded, prove lemmas that help the proof, where these lemmas themselves are guided by such high-level proof tactics. Needless to say, this is very hard for programmers to furnish, requiring them to not only understand the code and the specification language, but also the underlying proof mechanisms and tactics. Programmers with formal methods background, however, can typically take the verification through for simple properties of the software (see [26] and [30] for such practical case-studies).

The *natural proof technique*, proposed by Qiu et al. [32], suggests a way to alleviate this trigger/tactic annotation problem by identifying natural proof tactics for heap verification that are commonly used in manual proofs and deploying them automatically on code. Effective natural proofs need to have two properties (a) they should embody a set of tactics that are powerful enough to take the proofs of many verification tasks through, and (b) the search for proofs with the proof tactics must be efficient and predictable (preferably searchable using decidable logical theories and SMT solvers). In order to develop a set of tactics that is effective, the specification logic itself may need to be co-designed with natural proofs. Qiu et al. [32] provide a dialect of separation logic called DRYAD that forces the user to write specifications using primarily recursion (shunning unguarded quantification, and even tweaking the semantics of the separation logic mildly to ensure that it doesn't introduce arbitrary quantification), and develop a set of natural proof tactics that involve unfolding recursive definitions across the footprint of the program segment verified followed by an *uninterpreted* abstraction of recursive definitions. Furthermore, they encode these tactics using decidable SMT-solvable theories to build fast automatic proof techniques.

While the natural proof technique is very promising (Qiu et al. [32] describe 100+ data-structure programs automatically verified using their methodology), the tool they develop is only for a basic toy programming language consisting of heap manipulating commands. Consequently, it is not at all clear how the technique will fare in a realistic programming language, with a complex semantics and a complex memory model with a varied set of underlying types. In particular, handling the memory model itself often requires quantification, and it is highly unclear how the natural proof technique will work in such a setting.

In this paper, we build an automated deductive verification tool, called VCDRYAD for C programs against DRYAD separation logic specifications, where proofs are automated using natural proofs, developing the ideas in Qiu et al. [32] to a real programming language. The tool extends VCC, the C program verifier from Microsoft Research [9] by augmenting the deductive verification tool with DRYAD and natural proofs.

The technical novelty in our work is the *lifting* of natural proof techniques from the verification-condition-generation level (as suggested in Qiu et al. [32]) to the code-level: we automatically synthesize *annotations* at the code-level so that VCC interpreting these will automatically search for natural proofs of the properties it verifies. As we describe below, this involves several intricate mechanisms in order to (a) describe heaplets and separation logic semantics using a custom-defined theory of sets in VCC that exploit decidable array-theories, (b) synthesize annotations so as to place important consequences of destructive heap updates (including function calls) so that VCC can recover properties after such a destruction using the local reasoning provided by separation logic, and (c) careful ways to write precise annotations, sometimes directly writing at the underlying BOOGIE level, so as to side-step VCC's complex modeling of the C semantics and memory model into BOOGIE.

We now describe the architecture of the tool and the issues that motivated it, the challenges in following this architecture, some details of the synthesis of annotations, and the evaluation of the tool.

Architecture: Lifting Natural Proofs to the Code-Level. There are two choices for building natural proof tactics into the VCC verification pipeline, as depicted in Figure 1. The obvious architecture suggested by Qiu et al. [32] is the first architecture (Architecture A), shown on the left in Figure 1. Here, we encode natural proof tactics involving unfolding recursive definitions and their uninterpreted abstraction, and the encoding of heaplet semantics and specification into decidable theories, while generating the verification conditions. This has obvious advantages— we have complete control of the logical formulas being generated, and seeing the entire formula allows us to exploit natural proofs to the widest extent possible. However, this architecture is incredibly hard to engineer for a real language. The biggest problem is that the specifications, now in DRYAD separation logic and not native VCC specifications, will have to be weaved through the entire stack. VCC uses a fairly complex translation from C programs to BOOGIE, using a *typed object model* (rather than the official untyped bit-based memory model for C). This typed object model allows simpler reasoning on types, memory accesses, etc., to achieve better scalability, less annotation overhead, and greater versatility in proving well-written C programs correct [10]. Weaving the new definitions through all this would be considerably hard. We also lose the ability for VCC/BOOGIE to handle specifications written in their syntax, including possible triggers sent to the SMT solver, the interaction of the specification with the memory model, etc. unless we carefully transform these layers as well. Furthermore, the various logical models for handling the C semantics and memory model by VCC/BOOGIE itself introduce quantified constraints [10] that fall in undecidable theories and VCC augments these with automatic trigger mechanisms (hundreds of such quantified formulas are generated even for very simple programs, with hundreds of triggers). It's unclear how the natural proof VC generation will handle these constraints and orchestrate their interaction with the other recursive constraints. We believe that incorporating natural proofs at the lowest level will be hard in any verification stack (not just VCC) that handles a complex programming language.

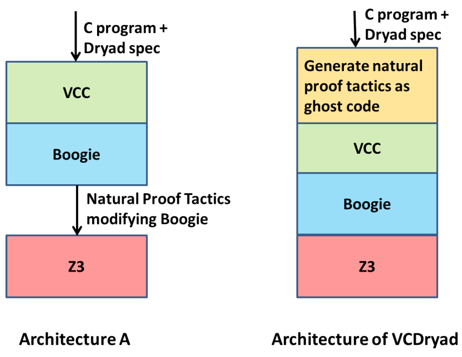

Figure 1: Choices of architecture for implementing natural proofs

The approach we follow in this paper is the second architecture depicted in Figure 1, where we engineer natural proofs at the *code-level*, writing annotations at the VCC input level that force natural proofs to be discovered down the pipeline. We lose the advantage of being able to control the precise verification conditions generated to the SMT solver. However, as we show in this paper, it is possible to do the translation of DRYAD specifications to first-order logic (using custom defined object sets to handle heaplet semantics and keep their manipulation within decidable theories) and encode the unfolding and abstractions of recursive definition tactics, all at the VCC level. We hence fully exploit the VCC/BOOGIE levels as black-boxes that manipulate our abstracted specifications, keeping our engineering entirely agnostic to the memory model handling and quantification trigger handling by these tools. While engineering the tool, we did occasionally look carefully at the Z3 constraints being generated and used this to formulate our VCC-level annotations in a way so as to ensure that the specification and reasoning that we add do not contribute to undecidability at the logic level, but this was minimal and the design is largely agnostic to the internals of VCC and BOOGIE.

Annotation Synthesis. Our tool VCDRYAD hence purely synthesizes annotations using the VCC syntax, at the level of the C code. The tool interprets the C program and the specification written in DRYAD, and performs three main tasks:

- Translates the separation logic DRYAD to VCC's roughly first order syntax. It models heaplets defined by recursive DRYAD formulas as recursively defined sets of objects, where sets of objects and set-operations on them are modeled using point-wise functions on arrays that are amenable to automatic reasoning at the SMT level. It performs these translations for definitions, annotations and assertions throughout the code.
- The recursive definitions, including the recursive definitions of heaplets, are then modeled as entirely uninterpreted functions. However, the recursive definitions are captured logically and their precise definition is unfolded at every point of dereference in the program, capturing the two natural proof tactics. Our tool hence inserts these expansions throughout the code, doing some minimal static analysis to figure the right definitions to expand (based on the type of the variable being dereferenced, etc.).
- The most intricate part of the annotations are for function-calls and statements that perform destructive heap updates (these two are handled very similarly). Whenever there is a function call, VCC havocs all information it currently has, and we restore all information that we can derive from the heaplet semantics of the separation logic contract for the function being called. Intuitively, the pre/post condition for the called function implicitly defines the heaplet that could be modified, and we explicitly restore the field-pointers and the values of recursive definitions that were not affected by the function called.

Note that the annotations that we add are far removed from the *triggers* that VCC programmers are encouraged to use to aid proofs, which are meant to help provide instantiations of quantified constraints in order to prove them unsatisfiable. We write no triggers at all. The natural proof method that we employ seems to be a very different technique that tries to tie the recursion in the specification with the recursion/iteration in the code in order to extract simple inductive proofs of correctness.

VCDRYAD is hence a tool for C programs manipulating data-structures where only annotations of pre/post conditions and loop invariants are required. The tool is designed to make most reasoning of data-structures completely automatic, without the use of programmer-provided tactics and lemmas. However, when a proof fails, the programmer can still see and interact with the DRYAD specifications, its translation to first-order logic, and the automatically generated tactics to help the proof go through, since all our effort explicitly resides at the code-level.

Evaluation. The proof for any sound but incomplete system is in the pudding. Our main hypothesis going into the evaluation was that we can make natural proofs work for a complex language like C by exerting enough control through annotations at the code level. We give evidence to this hypothesis by developing the tool and experimenting with it on more than 150 C programs manipulating data-structures.

Our program suite consists of standard routines for data structure manipulation (i.e., singly-linked, doubly-linked list, and trees), real world programs taken from well-known open source projects (i.e., Glib and OpenBSD), custom OS kernel data structures, Linux kernel routines used in a software verification competition [5], and programs used to evaluate approaches based on decidable separations logic fragments reported in Piskac et al. [31] and Itzhaky et al. [20].

The tool VCDRYAD was able to prove *all* the above programs correct, automatically, without any user-provided proof tactics. With this automation, VCDRYAD presents a significant advance in automated reasoning in deductive verification for heap manipulation. While there are, of course, several other high-level properties (such as properties about graphs which are not recursively definable) for which effective natural proofs are not known, we believe that for most programs manipulating standard inductively defined data-structures, verification using recursion can be significantly automated.

2. DRYAD and Natural Proofs

In this section, we give a brief description of the DRYAD logic and the natural proof technique. We present only those details that are required for understanding the encoding to VCC annotations presented in this paper. The interested reader may find a more complete and more formal specification of DRYAD in [32].

Syntax. DRYAD is a dialect of separation logic, and its syntax is given in Figure 2. It is a multi-sorted separation logic, supporting five sorts: location, integer, set/multiset of integers, and set of locations, and allows all standard operations over these sorts, as well as the separating conjunction from separation logic.

DRYAD disallows explicit quantification but allows user-provided recursive definitions, which in turn allow a form of guarded quantification. Inductively defined data structures can be defined naturally using such recursive definitions. Furthermore, DRYAD has a slightly different semantics from separation logic that ensures statically determined heaplets for recursive definitions.

There are several different kinds of terms in the logic— location terms, integer terms, "set of locations" terms, "set of integers" terms, and "multi-set of integers" terms; all except the first can use recursive definitions of the corresponding type. Formulas combine these terms in the usual ways: integers with respect to arithmetic relations, sets/multisets with respect to the $<$ (or $\leq$) relation, which checks whether all elements of the first set are less than (or equal to) all elements of the second set, membership in sets. Formulas are closed under conjunction and disjunction (negation needs to be pushed all the way in) and under the separation conjunction operator $*$.

We will consider as a running example binary search trees. Binary search trees can be defined recursively in DRYAD as:

$$\begin{aligned} bst_{\{l,r\}}(x) \quad \overset{def}{=} \quad & (x = \texttt{nil} \land \texttt{emp}) \quad \lor \\ & \big((x \overset{l,r,k}{\mapsto} \mathsf{left}, \mathsf{right}, \mathsf{m}) * \\ & (bst(\mathsf{left}) \ \land \ keys(\mathsf{left}) \leq m) * \\ & (bst(\mathsf{right}) \ \land \ \mathsf{m} \leq keys(\mathsf{right})) \ \big) \end{aligned}$$

In the definition above, $x \overset{l,r,k}{\mapsto} \mathsf{left}, \mathsf{right}, \mathsf{m}$ implicitly and guardedly quantifies $\mathsf{left}, \mathsf{right}, \mathsf{key}$ to denote the values of x's fields l, r and k; note that these are uniquely determined. The first use of $\leq$ in the above asserts that every integer in the left-hand set is at most the integer on the right; similar semantics holds for the second usage. The definition of *keys* is given by the following *if-then-else* term:

$$\begin{aligned} & keys_{\{l,r\}}(x) \overset{def}{=} \mathrm{ITE}\big(x = \texttt{nil} : \emptyset_I, \\ & (x \overset{l,r,k}{\mapsto} xl, xr, xk) : \ \ \{xk\} \cup keys(xl) \cup keys(xr)\big) \end{aligned}$$

Semantics. The semantics of DRYAD is consistent with standard separation logic for basic constants and connectives like `emp`, separating conjunction $*$, and other Boolean operations. However, DRYAD enforces an exact heaplet semantics for recursive definitions. Unlike standard separation logic, the heaplet for a recursive definition is uniquely determined— it always is the set of all locations reachable using certain field pointers, with possible *stopping* locations. These field pointers and stopping location terms are given syntactically in the definition.

For the definition of $bst_{\{l,r\}}(x)$, its heaplet is the set of all reachable locations from x via fields l and r, and can be recursively defined. As a naming convention, let the name of the heaplet corresponding to a definition d be d_$\mathsf{heaplet}$. Then the heaplet of bst

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

$$
\begin{array}{rrcl}
\mathit{Loc}\text{ Term:} & lt & ::= & x \mid \texttt{nil} \\
\mathit{Int}_L\text{ Term:} & it & ::= & c \mid j \mid i(lt) \mid it + it \mid it - it \\
\mathcal{S}(\mathit{Loc})\text{ Term:} & slt & ::= & \emptyset_L \mid L \mid \{lt\} \mid sl(lt) \mid slt \cup slt \mid slt \cap slt \mid slt \setminus slt \\
\mathcal{S}(\mathit{Int})\text{ Term:} & sit & ::= & \emptyset_I \mid S \mid \{it\} \mid si(lt) \mid sit \cup sit \mid sit \cap sit \mid sit \setminus sit \\
\mathcal{MS}(\mathit{Int})_L\text{ Term:} & msit & ::= & \emptyset_M \mid \mathit{MS} \mid \{it\}_m \mid msi(lt) \mid msit \cup msit \mid msit \cap msit \mid msit \setminus msit
\end{array}
$$

$$
\begin{array}{lll}
\text{Formula:}\ \varphi & ::= & \texttt{true} \mid \texttt{false} \mid q \mid p(lt) \mid \texttt{emp} \mid lt \overset{\vec{pf},\,\vec{df}}{\longmapsto} (\vec{lt}, \vec{it}) \mid lt = lt \mid lt \neq lt \mid it \leq it \mid it < it \mid sit \leq sit \mid sit < sit \\
& & \mid msit \leq msit \mid msit < msit \mid slt \subseteq slt \mid slt \not\subseteq slt \mid sit \subseteq sit \mid sit \not\subseteq sit \mid msit \sqsubseteq msit \mid msit \not\sqsubseteq msit \\
& & \mid lt \in slt \mid lt \notin slt \mid it \in sit \mid it \notin sit \mid it \in msit \mid it \notin msit \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi
\end{array}
$$

$$
\begin{array}{rcl}
\text{Recursive function :}\ f_{\vec{pf},\vec{v}}(x) & \overset{def}{=} & \mathrm{ITE}\Big(\varphi_1^f(x,\vec{v},\vec{s}) : t_1^f(x,\vec{s})\ ;\ \ldots\ ;\ \varphi_k^f(x,\vec{v},\vec{s}) : t_k^f(x,\vec{s})\ ;\ \text{default} : t_{k+1}^f(x,\vec{s})\Big) \\
\text{Recursive predicate :}\ p_{\vec{pf},\vec{v}}(x) & \overset{def}{=} & \varphi^p(x,\vec{v},\vec{s})
\end{array}
$$

Figure 2: Syntax of DRYAD

can be defined using the following recursive definition (in classical logic, with standard least fixed-point semantics):

$$
\begin{array}{l}
\mathit{bst}\_\mathsf{heaplet}(x) \overset{def}{=} \\
\quad \mathrm{ITE}\big(x = \texttt{nil} : \emptyset_L, \\
\quad\quad (x \overset{l,r,k}{\mapsto} xl, xr, xk) : \\
\quad\quad\quad \{x\} \cup \mathit{bst}\_\mathsf{heaplet}(xl) \cup \mathit{bst}\_\mathsf{heaplet}(xr)\big)
\end{array}
$$

Now to interpret the recursive definition $bst(x)$, on a heaplet with domain R, if R is exactly the set of locations described by the reach set ($\mathit{bst}\_\mathsf{heaplet}(x)$), then the semantics of *bst* is defined to be the *least fixed-point* that satisfies its definition. Otherwise $bst(x)$ is undefined.

The heaplet for *keys*, *keys*_heaplet, can be defined similarly. In general, for any recursive definition d in DRYAD, its corresponding heaplet definition $d\_\mathsf{heaplet}(x)$ *always* uniquely delineates the heap domain for $d(x)$. Note that the subscript $\{l, r\}$ indicates the fields via which the reachable heaplet is defined, but is usually omitted when clear from the context.

By deterministically delineating the heap domain for recursive definitions, DRYAD can syntactically determine the *domain-exactness* and the *scope* for any part of a DRYAD formula. Intuitively, the scope of a formula/term is the domain of the *minimum* heaplet required to interpret it; and a formula/term is domain-exact if it cannot be satisfied/evaluated when the heaplet domain is larger than the scope. Neither the scope nor the domain-exactness can be syntactically determined in standard separation logic. But in DRYAD, for each basic binary connective $t \sim t'$ (other than $*$, $\wedge$, $\vee$ and $\neg$),

$$
\begin{array}{rcl}
\text{domain-exact}(t \sim t') & \overset{def}{=} & \text{domain-exact}(t) \wedge \text{domain-exact}(t') \\
\text{scope}(t \sim t') & \overset{def}{=} & \text{scope}(t) \cup \text{scope}(t')
\end{array}
$$

Then when interpreted on a heaplet with domain R, the connective $\sim$ has the normal semantics if:

- either t or t' is not domain-exact, and $\text{scope}(t \sim t') \subseteq R$; or
- both t and t' are domain-exact, and there exists R_1, R_2 such that $R = R_1 \cup R_2$, and t/t' has a well-defined semantics on R_1/R_2, respectively.

Otherwise, $t \sim t'$ has no well-defined semantics.

Translation to Classical Logic. The determinacy of the heap domain for recursive definitions is important for exploiting translation to quantifier-free classical logic using the theory of sets. We can translate DRYAD to classical logic that models heaplets as sets of locations constrained appropriately. As the heap domain for a recursive definition is a recursively-defined heaplet, we can translate

```
BNode * bst_insert_rec(BNode * x, int k)
   requires bst(x) & (~ (k i-in keys(x))) ;
   ensures bst(ret)
   ensures keys(ret) s= (old(keys(x)) union (singleton k)))) ;
{
  if (x == NULL) {
    BNode * leaf = (BNode *) malloc(sizeof(BNode));
    _(assume leaf != NULL)
    leaf->key = k; leaf->l = NULL; leaf->r = NULL;
    return leaf;
  } else {
    BNode * xl = x->l; BNode * xr = x->r;
    if (k < x->key) {
      BNode * tmp = bst_insert_rec(xl, k); x->l = tmp;
      return x;
    } else {
      BNode * tmp = bst_insert_rec(xr, k); x->r = tmp;
      return x;
    } } }
```

Figure 3: Recursive Implementation of BST-Insertion

each recursive definition from DRYAD to classical logic with recursion. For example, the DRYAD predicate *bst* defined above can be translated to the following definition:

$$
\begin{array}{rcl}
\mathsf{bst}(x) & \overset{def}{=} & (x = \texttt{nil} \wedge \mathsf{bst\_heaplet(x)} = \emptyset_L) \ \vee \\
& & \big(\, x \neq \texttt{nil} \\
& & \ \wedge (x \notin \mathsf{bst\_heaplet}(x.l) \cup \mathsf{bst\_heaplet}(x.r)) \\
& & \ \wedge (\mathsf{bst}(x.l)\ \wedge\ \mathsf{keys}(x.l) \leq x.k) \\
& & \ \wedge (\mathsf{bst}(x.r)\ \wedge\ x.k \leq \mathsf{keys}(x.r)) \\
& & \ \wedge \backslash\mathsf{disjoint}(\mathsf{bst\_heaplet}(x.l), \mathsf{bst\_heaplet}(x.r)) \\
& & \ \wedge \mathsf{bst\_heaplet}(x) = \\
& & \quad\quad \{x\} \cup \mathsf{bst\_heaplet}(x.l) \cup \mathsf{bst\_heaplet}(x.r)\ \big)
\end{array}
$$

A similar translation can be done from *bst*_heaplet, *keys*, and *keys*_heaplet.

Natural Proofs. Natural proofs and DRYAD have been co-designed, where proofs exploit the purely recursive formulation provided in the logic, with no explicit quantification (DRYAD allows implicit quantification, but these are always *guarded*, and hence is uniquely determined in the context). Natural proofs are sound but incomplete proof tactics that work for many programs, and are derived from common tactics found in manual proofs. In particular, the work in [32] identifies two main tactics (see also previous work [25] and [35] where these tactics were studied earlier). The tactics are to (a) unfold recursive definitions across the footprint (the locations explicitly dereferenced in the program) of the program segment being verified, and (b) to make the recursive functions uninterpreted. Intuitively, (b) causes too much loss of precision that (a) recovers by making sure that the semantics of recursive definitions are at least correctly described on the locations on the footprint. Natural proofs tend to find pure induction proofs— if a program segment calls a function f, then the natural proof would apply the pre/post condition of f to infer certain facts hold for recursive definitions when f returns, but will not unravel these definitions further, hence looking for a simple induction proof.

Consider the example of insertion into a binary search tree presented in Figure 3, with pre/post conditions written in DRYAD. After translating the pre/post conditions to classical logic (elaborated in Section 3), each Hoare-triple extracted from the program corresponds to a verification condition, which is a formula in classical logic with recursively defined bst, keys, etc. For example, the following formula is the verification condition for the case of inserting k into the left subtree of x (proving only the BST-ness is preserved):

$$\Big[\ \mathsf{bst}(x) \wedge k \notin \mathsf{keys}(x) \wedge x \neq \mathsf{NULL} \wedge k < key(x) \wedge \\ \Big(\mathsf{bst}(l(x)) \wedge k \notin \mathsf{keys}(l(x))\ ) \rightarrow \\ (\ \mathsf{bst}(tmp) \wedge \mathsf{keys}(tmp) = \mathsf{keys}(l(x)) \cup \{k\}\Big) \wedge \\ l'(x) = tmp \wedge r'(x) = r(x)\ \Big] \ \longrightarrow\ \mathsf{bst}'(x)$$

where bst' is the updated version of bst, defined using updated fields (l' and r'). Now we can prove it using the natural proof strategy. For this example, the footprint is simply x, the only dereferenced variable. Therefore, we unfold *all* recursive definitions, namely bst, keys, bst' and keys', and their corresponding heaplet definitions, on x. For instance, every occurrence of $\mathsf{bst}(x)$ is replaced with its unfolded definition presented before. Once we make those recursive definitions uninterpreted, the validity of the formula can be easily proved by a theorem prover supporting the theory of sets.

Qiu et al. [32] show that in many data-structure manipulating programs, the inductive invariant can often be used without further unfolding to prove the program correct, and hence natural proofs often exist. Moreover, the validity of the produced formulas is decidable by SMT solvers supporting the array property fragment [14]. This approach looks for natural proofs by encoding verification conditions appropriately, but we will be concerned about encoding natural proofs at the code-annotations-level, which is the main technical contribution of this paper.

3. Translating DRYAD Specifications and Modeling Natural Proofs as Ghost Annotations

In this section, we describe in detail the way we leverage natural proofs for C programs against DRYAD specifications by encoding natural proof tactics automatically into carefully crafted ghost-code annotations. The ghost code is directly at the source code level, handled by VCC, and consists of first order annotations that fall into decidable theories handled by any standard SMT solver. These automatically generated annotations help VCC carry out an automatic natural proof of the C program, freeing the programmer from guiding proofs using proof tactics.

We describe the synthesis in three phases. The first is on translating recursive definitions to first-order VCC annotations capturing the definitions as both uninterpreted functions as well as defining unfoldings of them according to their recursive definitions. The second phase describes how DRYAD annotations in code (pre/post conditions, loop invariants, assertions, etc.) are translated to VCC specifications. Finally, in the third phase, which is the most complex, we encode natural proof tactics using ghost annotations, unfolding recursive definitions on footprints and preserving the heap that doesn't change across statements and functions that modify the heap.

3.1 Phase I: Translating recursive definitions

As VCC and BOOGIE specifications have to be written in classical logic, roughly first-order[1], we need to translate DRYAD separation logic specifications to classical logic.

The first step is to translate recursive definitions. As we mentioned in the previous section, each recursive definition d in DRYAD can be translated to two recursive definitions in classical logic: one is the classical-logic equivalent of d, the other one recursively defines the heap domain for d, namely d_heaplet, which is the reachable locations according to certain pointer fields. In this work, these recursive definitions in classical logic are not directly amenable for developing a BOOGIE/Z3-based verifier. Remember that our goal is to encode the natural proof tactics completely using VCC annotations and give up controlling the BOOGIE-level VC-generation. Therefore, to deploy the unfoldings and the formula abstraction at the VCC-level, we translate DRYAD definitions slightly differently. All the recursive definitions are now translated to *uninterpreted functions*. In addition we define predicates in VCC describing how to unfold the definitions at particular locations using their true recursive definitions.

As an example, for the DRYAD predicate bst which is recursively defined as in Section 2, we can translate it to bst and bst_heaplet, both uninterpreted, and define how they should be unfolded:

```
_(pure \bool bst(struct node * hd) _(reads \universe()) ;)
_(pure \oset bst_heaplet(struct node * hd) _(reads \universe()) ;)
_(pure \bool unfold_bst(struct node * hd) _(reads \universe())
  _(ensures \result == (bst(hd) ==
     ((hd == NULL && bst_heaplet(hd) == {})
     ||(hd != NULL
       && (bst_heaplet(hd) == {hd} \union bst_heaplet(hd->l)
                                   \union bst_heaplet(hd->r))
       && bst(hd->l) && bst(hd->r)
       && \intset_lt_set(keys(hd->r), hd->key)
       && \intset_lt_set(hd->key, keys(hd->r))
       && \disjoint(bst_heaplet(hd->l), bst_heaplet(hd->r))
       && !(x \in bst_heaplet(hd->l)
                         \union bst_heaplet(hd->r))))));)
_(pure \bool unfold_bst_heaplet(struct node * hd)
    _(reads \universe())
    _(ensures \result == (
        (hd == NULL && bst_heaplet(hd) == {}) ||
        (hd != NULL && bst_heaplet(hd) ==
            ({hd} \union bst_heaplet(hd->l)
                  \union bst_heaplet(hd->r))))) ;)
```

The predicates unfold_bst and unfold_bst_healpet mimic the unfoldings of the recursive definitions bst and bst_healpet, respectively. When they are asserted on a location n, they guarantee that bst(n) and bst_heaplet(n) can be constructed from the evaluations of these functions on its neighbors, namely bst(n.l), bst_heaplet(n.l), bst(n.r), and bst_heaplet(n.r).

3.2 Phase II: Translating logical specifications

Using the determined heaplet semantics of DRYAD, Qiu et al. [32] show that DRYAD logic formulas can be translated to classical logic, still quantifier-free, preserving the heaplet semantics with respect to a given heaplet H. We follow a similar recursive translation as in [32], but adapt it to VCC syntax and the translation of recursive definitions presented above. Figure 4 shows this translation. We

[1] The logic is typically first-order logic but over a richer class including maps, sets, arrays, etc., and further allows pure recursive functions described in FOL as well.

$$
\begin{array}{rcl}
T_{VCC}(\mathit{var}\ /\ \mathit{const},\ \mathsf{G}) & \equiv & \mathit{var}\ /\ \mathit{const} \\
T_{VCC}(\{t\}\ /\ \{t\}_m,\ \mathsf{G}) & \equiv & \{t\}\ /\ \{t\}_m \\
T_{VCC}(t\ \text{op}\ t',\ \mathsf{G}) & \equiv & T_{VCC}(t,\mathsf{G})\ \text{op}\ T_{VCC}(t',\mathsf{G}) \\
T_{VCC}(f(lt),\ \mathsf{G}) & \equiv & \text{ITE}\ \big(\mathsf{f\_heaplet}(lt) = \mathsf{G},\ \mathsf{f}(lt),\ \mathtt{undef}\big) \\
T_{VCC}(\mathtt{true}\ /\ \mathtt{false},\ \mathsf{G}) & \equiv & \mathtt{true}\ /\ \mathtt{false} \\
T_{VCC}(\mathtt{emp},\ \mathsf{G}) & \equiv & \mathsf{G} = \{\} \\
T_{VCC}(lt \xmapsto{\vec{pf},\vec{df}} (\vec{lt},\vec{it}),\ \mathsf{G}) & \equiv & \mathsf{G} = \{lt\} \\
 & & \wedge \bigwedge_{pf_i}\ T_{VCC}(lt,\mathsf{G}).pf_i = T_{VCC}(lt_i,\mathsf{G}) \\
 & & \wedge \bigwedge_{df_i}\ T_{VCC}(lt,\mathsf{G}).df_i = T_{VCC}(it_i,\mathsf{G}) \\
T_{VCC}(p(lt),\ \mathsf{G}) & \equiv & \mathsf{p}(lt) \wedge \mathsf{G} = \mathsf{p\_heaplet}(lt) \\
T_{VCC}(t \sim t',\ \mathsf{G}) & \equiv & \begin{cases} t \sim t' & \text{if } t \sim t' \text{ is not domain-exact} \\ t \sim t' \wedge \mathsf{G} = \text{scope}(t \sim t') & \text{otherwise} \end{cases} \\
T_{VCC}(\varphi \wedge \varphi',\ \mathsf{G}) & \equiv & T_{VCC}(\varphi,\mathsf{G}) \wedge T_{VCC}(\varphi',\mathsf{G}) \\
T_{VCC}(\varphi \vee \varphi',\ \mathsf{G}) & \equiv & T_{VCC}(\varphi,\mathsf{G}) \vee T_{VCC}(\varphi',\mathsf{G})
\end{array}
$$

$$
T_{VCC}(\varphi * \varphi',\ \mathsf{G}) \equiv \begin{cases}
T_{VCC}\big(\varphi,\ \text{scope}(\varphi)\big)\ \wedge\ T_{VCC}\big(\varphi',\ \text{scope}(\varphi')\big) \\
\quad \wedge\ \text{scope}(\varphi) \cup \text{scope}(\varphi') = \mathsf{G} \\
\quad \wedge\ \text{scope}(\varphi) \cap \text{scope}(\varphi') = \emptyset \\
\qquad \text{if both } \varphi \text{ and } \varphi' \text{ are domain-exact} \\[1ex]
\text{scope}(\varphi) \subseteq \mathsf{G}\ \wedge\ T_{VCC}\big(\varphi,\ \text{scope}(\varphi)\big)\ \wedge \\
T_{VCC}\big(\varphi',\ \mathsf{G} \setminus \text{scope}(\varphi)\big) \\
\qquad \text{if only } \varphi \text{ is domain-exact} \\[1ex]
\text{scope}(\varphi') \subseteq \mathsf{G}\ \wedge\ T_{VCC}\big(\varphi',\ \text{scope}(\varphi')\big)\ \wedge \\
T_{VCC}\big(\varphi,\ \mathsf{G} \setminus \text{scope}(\varphi')\big) \\
\qquad \text{if only } \varphi' \text{ is domain-exact} \\[1ex]
T_{VCC}\big(\varphi,\ \text{scope}(\varphi)\big)\ \wedge\ T_{VCC}\big(\varphi',\ \text{scope}(\varphi')\big) \\
\quad \wedge\ \text{scope}(\varphi) \cup \text{scope}(\varphi') \subseteq \mathsf{G} \\
\quad \wedge\ \text{scope}(\varphi) \cap \text{scope}(\varphi') = \texttt{\textbackslash emptyset} \\
\qquad \text{if neither } \varphi \text{ nor } \varphi' \text{ is domain-exact}
\end{cases}
$$

Figure 4: Translating DRYAD specs to VCC specs
(with respect to a heaplet G)

omit the formal definition for the scope function, which intuitively is the *minimum* heap domain required to evaluate the formula/expression (see [32] for details). Given a pre-/post-condition or a loop invariant φ in DRYAD, we introduce a ghost set variable G in the translation, denoting the local heaplet manipulated by the current function, and replace φ with $T_{VCC}(\varphi, \mathsf{G})$. For each function/method m in the program, we assume its pre-/post-conditions are domain-exact, i.e., we can statically compute the required heaplet, namely $\mathsf{G\_pre\_m}(\vec{p})$ and $\mathsf{G\_post\_m}(\mathsf{ret},\ \vec{p})$, where $\vec{p}$ is the location parameters of m. Let the precondition of m be $\varphi_{\mathsf{pre-m}}(\vec{p})$ and let the postcondition of m be $\varphi_{\mathsf{post-m}}(\mathsf{ret},\ \vec{p})$; then in VCC, we translate them to the precondition $T_{VCC}(\varphi_{\mathsf{pre\text{-}m}}(\vec{p}), \mathsf{G\_pre\_m}(\vec{p}))$ and postcondition $T_{VCC}(\varphi_{\mathsf{post\text{-}m}}(\mathsf{ret},\ \vec{p}), \mathsf{G\_post\_m}(\mathsf{ret}, \vec{p}))$.

The translation of post-conditions is a bit more tricky, as the postcondition can refer to expressions or subformulas that are evaluated on the state *before* the function call, by using the \old() function (such old-state lookups are not allowed in [32]). In our translation, old expressions/formulas are translated in a *heapless* manner, i.e., the heaplet with formulas involving the pre-state are considered to be empty. This, in essence, is equivalent to having some properties of the pre-state recorded in auxiliary ghost variables that are assigned at the beginning of the function, and using them in the post-condition (where they will be heapless, since they are state variables).

The current heaplet G is properly maintained using ghost-variable updates during the program execution (see Section 3.3 for details on how this is done).

Consider again the bst insertion algorithm presented in Figure 3. Note that there are no proof tactics provided by the user (in contrast, the verification of such a routine in a tool like VeriFast would contain a large number of proof tactics and lemmas tying the implementation to the specification [2]).

Using the translation we just described, the precondition gets translated in VCDRYAD to the VCC precondition:

```
_(requires bst(x) && !\intset_in(k, \in keys(x)))
_(requires bst_heaplet(x) == keys_heaplet(x))
_(requires G == bst_heaplet(x))
```

and the postcondition gets translated to:

```
_(ensures bst(\result)
_(ensures keys(\result) == \intset_union(\old(keys(x)), k)))
_(ensures G == bst_heaplet(\result))
_(ensures bst_heaplet(\result) = keys_heaplet(\result))
```

3.3 Phase III: Natural Proofs for VCC

We now describe the core of the annotation synthesis for forcing VCC to find natural proofs. Our annotation synthesizer simply instruments ghost code before and after each statement of the program, with some help of some (local) static analysis to figure out the parameters for the instrumentation. Figure 5 gives the precise ghost code instrumented before and/or after each statement.

The ghost code for each statement does four main things to encode inferable facts from the program state resulting from the execution of the statement:

Unfolding: Assumptions that the unfolding of recursive definitions hold at certain footprint locations;

Preservation: Inferences that fields and recursive definitions on locations that are maintained across destructive updates and function calls;

Current heap update: Updating the current heap maintained by the ghost variable G, after memory allocations and function calls;

State memoization: Create ghost variables to remember the state of the program at various points so that annotations can refer back to these states later to update information on locations pointed to by program variables at this state.

The current heaplet G is maintained as the domain of the heap changes. Since the current heaplet changes only with a `malloc`, `free` or a function call, we instrument different statements for each case. We instrument $\mathsf{G} = \mathsf{G} \cup \{x\}$ after each statement of the form $\mathsf{x} = \mathsf{malloc}()$, and we instrument $\mathsf{G} = \mathsf{G} \setminus \{x\}$ after each statement of the form free(x). After a function call to m, G can be updated by excluding G_m_pre and then including G_m_post: $\mathsf{G} = (\mathsf{G} \setminus \mathsf{G\_m\_pre}) \cup \mathsf{G\_m\_post}$. Note that when a DRYAD loop invariant is translated to a classical-logic formula with respect to current heaplet variable G, the obtained formula is still a valid loop invariant, as the current heap G is maintained using appropriate ghost updates through each iteration.

In order to simplify presentation, we will assume that in the program, location dereferences only appear in the form of $\mathsf{u} = \mathsf{v.f}$ or $\mathsf{u.f} = \mathsf{v}$; all other statements (including conditions) with dereferences can be split into simpler ones, for example, the statement $\mathsf{u.f.g} = \mathsf{v}$ can be split into two sequential statements: $\mathsf{tmp} = \mathsf{u.f}$; $\mathsf{tmp.g} = \mathsf{v}$.

We show in Figure 5 the ghost code instrumented for each basic statement. More complex statements such as if..then..else.., while, etc., are not transformed, i.e., they remain as they are, with the leaf statements transformed using the table.

The instrumentation relies on two sets of location variables: $FP(i)$ and $EFP(i)$, assuming i is the position of the current pro-

[2] `http://people.cs.kuleuven.be/~bart.jacobs/verifast/examples/sorted_bintree.c.html`

Stmt	After Instrumentation	Comment
u = $\langle expr \rangle$	u = $\langle expr \rangle$;	

u = v.f
(assume v is of type T)
	assume $\bigwedge_{d \in defs(T)} ($
	_dryad_unfold_d(v) $\wedge$ _dryad_unfold_d_heaplet(v)$)$;
	_(ghost T * _dryad_fp_$\langle i \rangle$ = v;)
	_(ghost \oset _dryad_scope_$\langle i \rangle$ = allLocFields(v);)
	
	u = v.f;
		
		(Unfold dereferenced loc)
		(State memoization)
		(State memoization)

u = malloc()
(assume u is of type T)
	u = malloc()
	
	_(ghost G = \oset_union(G, {u});)
		
		(Current heaplet update)

free(u)
(assume u is of type T)
	free(u)
	
	_(ghost G = \oset_diff(G, {u});)
		
		(Current heaplet update)

u.f = v (assume u is of type T)	assume $\bigwedge_{d \in defs(T)} ($ _dryad_unfold_d(u) $\wedge$ _dryad_unfold_d_heaplet(u)$)$; _(ghost T * _dryad_fp_$\langle i \rangle$ = u;) _(ghost \oset _dryad_scope_$\langle i \rangle$ = allLocFields(u);) _(ghost \state _dryad_S_$\langle i \rangle$ = \now();) u.f = v; assume $\bigwedge_{p \in FP(i)} \bigwedge_{d \in defs(p)} ($ _dryad_unfold_d(p) $\wedge$ _dryad_unfold_d_heaplet(p)$)$; assume $\bigwedge_{p \in EFP(i)} \bigwedge_{d \in defs(p)} \Big($ (! \oset_in(u, d_heaplet(p))) ==> (d(p) == \at(_dryad_S_$\langle i \rangle$, d(p)) && d_heaplet(p) == \at(_dryad_S_$\langle i \rangle$, d_heaplet(p))) $\Big)$;	 (Unfold dereferenced loc) (State memoization) (State memoization) (State memoization) (Unfolding on entire footprint) (Preserving definitions)
u = m($\vec{v}$, $\vec{z}$) (assume $\vec{v}$ is location parameters and $\vec{z}$ is int parameters)	_(ghost \state _dryad_S_$\langle i \rangle$ = \now();) u = m($\vec{v}$, $\vec{z}$); assume $\bigwedge_{p \in FP(i)} \bigwedge_{d \in defs(T_p)} ($ _dryad_unfold_d(p) $\wedge$ _dryad_unfold_d_heaplet(p)$)$; assume $\bigwedge_{p \in EFP(i)} \bigwedge_{d \in defs(T_p)} \Big($ \oset_disjoint(G_pre_m($\vec{v}$), d_heaplet(p)) ==> (d(p) == \at(_dryad_S_$\langle i \rangle$, d(p)) && d_heaplet(p) == \at(_dryad_S_$\langle i \rangle$, d_heaplet(p))) $\Big)$; assume $\bigwedge_{p \in FP(i)} \bigwedge_{f \in flds(T_p)} \Big($ (! \oset_in(p, G_pre_m($\vec{v}$))) ==> \at(_dryad_S_$\langle i \rangle$, p.f) == p.f$\Big)$; _(ghost G = \oset_union(\oset_diff(G, G_pre_m($\vec{v}$)), G_post_m(u));)	(Current state memoization) (Unfold on entire footprint) (Preserving definitions) (Preserving fields) (Current heaplet update)

Figure 5: Ghost code instrumented for every statement
(*assuming the current statement is at position i in the program*)

gram statement (program counter). Intuitively, *FP*(i) is the *footprint*, the set of dereferenced location variables, possibly memoized in previous locations using ghost variables. *EFP*(i) is the *extended footprint* which includes all location variables in scope (where variables at other locations are captured using ghost-variables) as well as the ghost-variables that capture the locations obtained by dereferencing using location fields from these variables in the program. Variables in *FP*(i) are of the form _dryad_fp_$\langle j \rangle$. Whenever a dereference u.f appears in a statement at position j, there is a ghost variable _dryad_fp_$\langle j \rangle$ defined before the dereferencing, remembering where u points to; and there is a ghost set variable _dryad_scope_$\langle i \rangle$ that remembers all location fields of the dereferenced variable. Then _dryad_fp_$\langle j \rangle$ and _dryad_scope_$\langle j \rangle$ will be added to *FP*(i) and *EFP*(i), respectively, as long as it is visible at the statement i. Both $FP(i)$ and $EFP(i)$ are computed by a simple local static analysis of the program before the instrumentation, and we omit the details of their computation in this paper.

We now briefly describe what gets instrumented for each type of statement, following Figure 5. For statements that do not update the heap ($\mathsf{u} = \langle expr \rangle$ where $\langle expr \rangle$ is an expression without dereferencing), no instrumentation is done. For dynamic memory allocation (malloc() and free()), we only update the current heap variable G by including or excluding the location allocated or freed, respectively.

For a field lookup ($\mathsf{u} = \mathsf{v.f}$), we unfold all pertinent recursive definitions d and unfold the recursive definition corresponding to the heaplet of d (d_heaplet) defined on u before the lookup. Moreover, the location pointed to by v is stored in a ghost pointer associated with the current program location i (dryad_fp_i), to remember that this location is part of the footprint. The locations pointed to by the different fields from the dereferenced location v are also stored in a ghost variable (of type *object set*) pertaining to the current location (_dryad_scope_i). These field pointers are remembered so that we can restore them later after a destructive update.

Instrumentation for destructive updates ($\mathsf{u.f} = \mathsf{v}$) is more complex. We first unfold all pertinent recursive definitions for the location pointed to by u (since the heap is updated by the statement). We also store the current location u and the locations pointed to by its fields in ghost variables, as in the previous case. Then, after the statement, we unfold all recursive definitions on all locations in the footprint. The set $FP(i)$ statically captures the set of variables of the form dryad_fp_j, for various program locations in scope. We unfold recursive definitions on all of these locations *in the current program state*. Finally, we also explicitly infer the fact that for any location p in scope and any recursive definition d, if the heaplet corresponding to the definition (d_heaplet(p)) does not contain u, both the heaplet and the definition itself remain unchanged after the update.

Function calls are handled similar to destructive updates, but with a more complex inference after the call. First, we explicitly infer the recursive definitions that are maintained across the function call when the heaplet of the definition and the heaplet modified by the function (G_pre_m($\vec{\mathsf{v}}$)) are disjoint. Secondly, as VCC havocs the entire heap after a function call (since the modified set is unspecified), we need to explicitly infer that the field pointers from a location in the footprint p.f is unchanged whenever p is disjoint from the heaplet (G_pre_m($\vec{\mathsf{v}}$)) the function call modifies. Finally, the current heap variable G also gets updated (removing the heaplet of the pre-condition and adding the heaplet of the post-condition).

Regarding the soundness of the assumption annotations that we introduce, note that *unfold* definitions simply declare the recursive definition for a location and hence are always sound, and our tool liberally strews them across all possible locations touched by the program. The *preservation* assumptions, on the other hand, crucially rely on the heaplet semantics of DRYAD, and have to be instrumented carefully.

4. Design and Implementation of VCDRYAD

We engineered our tool VCDRYAD [3] as an extension to the (open-source) deductive verification tool VCC, but restricted to sequential C programs. The tool essentially processes C programs with DRYAD specifications, translating the specifications to first-order ghost-code as described in the Section 3 and Figure 5. Currently VCDRYAD does not handle all complexities of the C language, in particular we forbid function pointers and pointer arithmetic. The effort in engineering the tool was about 1 person year, where most time was spent in building the precise ghost-code that we needed by inserting it both at the VCC level and to the BOOGIE level (the latter is done through VCC's macros that provide injection to the BOOGIE level). The tool is written in F#, extending the VCC transformers written in the same language.

4.1 Encoding sets/multisets of integers and locations

Lifting natural proof tactics from the verification-condition-level to the code-level calls for careful encoding of integer and location *sets*. Our first attempt was to encode integer and location sets (in particular, heaplets) using VCC's specification primitives and reusing existing object set definitions. In particular, the integer sets/multisets were encoded using VCC's map, with set operations described using lambda functions over the maps. VCC creates a custom axiomatization of these maps even though it translates VCs to BOOGIE language which itself provides support for maps. However, we were not able to prove even simple properties without adding additional axioms describing basic properties of sets.

After several verification attempts and by examining the output that VCC/BOOGIE provided to Z3, we decided to abandon the above approach, and instead proceeded to model integer and location sets differently. Integer sets and heaplets in DRYAD can be modeled in a decidable theory, as described in Qiu et al [32], using particular custom maps and the decidable array property fragment. Working at the VC level, the work in [32] had full control over the formulas passed to the underlying SMT solver, and hence could perform this encoding easily. We proceeded to model sets of locations using the type (`\objset`), which is used in VCC to handle object ownership. However, to remove the source of incompleteness mentioned above, we encoded sets of type T as Arrays from T to Boolean using Z3's extended theory of arrays [14]. By carefully examining the Z3 output, we removed all forms of universal quantification that emanated from our annotations by modeling in the above decidable theory and in the decidable array property fragment theory [6]. VCC/BOOGIE itself generates universally quantified formulas to capture C's memory model— in practice we found that the triggers provided by VCC/BOOGIE handled this part well enough, and so we let those be. We implemented the encoding of our sets at the BOOGIE level, using stubs provided in VCC's header prelude file.

4.2 DRYAD Type Invariants

Each DRYAD definition can be thought as a type (object) invariant. The notion of object invariants is typically associated with strongly typed object-oriented languages. Strong typing disallows distinct objects and fields to overlap. In C, however, the notion of object is weaker, it means that type definitions provide a way of interpreting a chunk of memory. This means that objects in C can overlap almost arbitrarily. Therefore, sound verification of C programs typically requires an untyped memory model, which unfortunately comes with high performance and annotation overhead. VCC provides a sound and efficient typed memory model for C by maintaining object validity through invariants and proof obligations that guarantee that objects do not alias. In VCC aliasing is achieved through ownership annotations referring to memory regions [9]. The handling of aliasing is the key difference between our approach and that of VCC. Instead of the ownership methodology, we use separation logic to describe object disjointness and employ natural proof tactics through ghost code to enable automated reasoning (as described in Section 3). However, we do leverage all the reasoning VCC performs to ensure object validity that does not rely on ownership specifications.

To integrate our approach within the VCC framework we derive DRYAD type invariants from translation of recursive DRYAD specifications to first-order logic VCC specifications (see Sec. 3.1). Given this translation, we need to extract suitable information to

[3] `http://www.cs.illinois.edu/~madhu/vcdryad`

perform unfolding and preserving definitions across the destructive updates and function calls. From the translated DRYAD definitions we have to extract information describing data structure invariants (such as singly-linked list, binary search tree, etc.), and its footprint definitions. We use a light-weight static analysis to find which DRYAD predicates and functions are associated with data structure definitions (expressed using `struct`), and the associated DRYAD predicates and their heaplet definitions. Moreover, we perform static analysis to determine fields of a data structure on which DRYAD specifications depend and vice versa. We use that information when performing the unfolding and preserving DRYAD definitions and field pointers as described in Figure 5 in Section 3.

4.3 Axioms relating recursive definitions

We follow the natural proof methodology [32] in adding several axioms that relate different recursive definitions for data-structures, including those that relate partial data-structures to complete ones (like list segments to lists) and those that unfold recursive definitions in the reverse direction (like unfolding linked list segments from the tail to the left). Note that these axioms are provided for each *class* of data-structures (like linked list segments, doubly linked lists, etc.) but are not specific to the program being verified. The success of automatic verification does crucially depend on these axioms, and automating these axioms would be interesting future work.

4.4 Debugging Unsuccessful Verification Attempts

The VCC/BOOGIE/Z3 pipeline being the underlying framework for our verifier provides additional help in the process of understanding unsuccessful verification attempts. The reasons for failed attempts are typically due to mistakes in the code or in the specifications (it is easy to write invariants that are correct but not inductive). When VCC reports that the property did not verify it means that Z3 could not discharge the VC and it produces a counterexample model. The counterexample model can be interpreted with VCC's BOOGIE Verification Debugger (BVD) [23], which can be useful in debugging the failed attempts. Z3 Inspector is also a useful tool that shows the precise properties the underlying solver is trying to prove at the code level. Our experience in writing and specifying 150 programs included several attempts where we wrote a wrong program or specification, but where these tools helped us find and debug them.

5. Evaluation

We evaluated our tool VCDRYAD on more than 150 data-structure manipulating routines [4], which in turn exercised the natural proof technique for C for thousands of verification conditions. The programs were written with user-defined data structure definitions and annotated with preconditions, postconditions, and loop invariants. No further proof tactics were provided.

VCDRYAD handled *all* our programs automatically. Table 1 shows the result of the experiments. We follow the naming convention that routines with suffix `rec/iter` denote recursive/iterative implementations. Our routines include standard manipulations of singly-linked, doubly-linked, circular lists, binary trees, AVL trees, etc. Some of these structures are hard to define recursively (e.g., doubly-linked and circular lists), and also difficult to verify inductively. Furthermore, these routines were verified for full functional correctness, including properties involving the precise set of keys modified, the balancing of trees, etc.

We used our tool to verify routines taken from various real world programs. In particular, we verified a large set of routines manipulating singly-linked and doubly-linked lists from a well-known Glib C library, and queue manipulations as implemented in the OpenBSD operating system. Furthermore, we verified custom data structure procedures developed in ExpressOS, an OS that uses formal verification to provide stronger security guarantees. Our set of benchmarks also includes some heap manipulation programs from a software verification competition (SV-COMP) [5]. Finally we verified the programs used to evaluate recent tools that handle weak but decidable fragments of separation logic, namely GRASShopper [31] and AFWP [20].

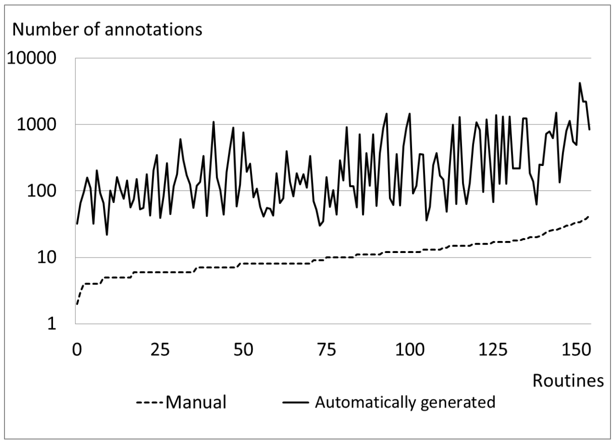

Figure 6: Comparison of manual annotation versus automatically generated annotations.

Figure 6 shows the number of manual and automatically generated annotations for the various routines, sorted in ascending order of the manual annotations they had. Note that y-axis, depicting number of annotations, is in a logarithmic scale. The tool adds 3X to 150X new annotations over the number of manual annotations ($\sim 30X$ in average). It is interesting to see that though we add lots of annotations (up to 4000 annotations for a routine), these annotations are in a much simpler theory (uninterpreted functions, arithmetic, plus decidable quantified theories of arrays), and actually helps the prover. Also, note that the annotations that we add are either assumptions or ghost variable updates, and hence the *number* of verification conditions do not increase. Manually provided proof tactics commonly used span a wide range, and natural proofs seem to be a good *uniform* way to discover most proofs automatically.

6. Related Work

The literature on verification of data-structure manipulation in programs is rich. We focus mainly on related work on logics and deductive verification techniques for data-structures. Separation logic [29, 33] has emerged as a succinct heap logic that is amenable to local reasoning. Decidable fragments of separation logics were identified first in [2, 3], and the tool SMALLFOOT provides automated reasoning for a restricted class of linked lists and trees. Decidable fragments of separation logic have been investigated in several recent works (see [13, 18, 20, 28, 31]) and a recent result for a considerably more expressive logic for bounded tree width structures is known [19] (though its practical applicability is unclear as it reduces reasoning to MSO reasoning). Non-separation logics that are decidable are also known [25, 27, 29].

Suter et al. [35, 36] explore proof tactics to prove properties of algebraic data-structures manipulated by functional programs. The work on natural proofs [25, 32] extend such tactics to imperative programs, where the proof tactics unfold recursive definitions across the footprint manipulated by imperative code and use for-

[4] `http://www.cs.illinois.edu/~madhu/vcdryad/examples/`

Benchmark and total LOC	Routine	Time (s) / Routine	Benchmark and total LOC	Routine	Time (s) / Routine
Singly-linked list 130 LOC	insert_front, copy_rec, insert_back_rec, append_rec, find_rec, reverse_iter, delete_all_rec	< 1	Sorted list 260 LOC	find_rec, find_last, insert_sort_rec, delete_all_rec, reverse_iter	< 1
Doubly-linked List 120 LOC	insert_front, insert_back_rec, append_rec, mid_insert, delete_all, mid_delete, meld	< 1		insert_iter	1
				concat_sorted	3
				merge_rec	8
Circular list 110 LOC	insert_front, insert_back_rec, delete_front, delete_back_rec	< 1		quick_sort_iter	6
				insert_sort_iter	20
BST 140 LOC	find_rec, find_iter, delete_rec	< 1	Treap 170 LOC	find_rec	< 1
				delete_rec	2
	insert_rec	1		insert_rec	10
	remove_root_rec	1		remove_root_rec	35
AVL-tree 320 LOC	leftmost_rec	< 1	Tree Traversals 100 LOC	preorder_rec, inorder_rec, postorder_rec	< 1
	avl_insert	4			
	avl_delete	20		inorder_tree_to_list_rec	3
	avl_balance	260			
glib/gslist.c Singly-Linked list 550 LOC	free, find, prepend, last, concat, append, insert_at_pos, insert_before, remove, remove_link, delete_link, reverse, nth, nth_data, position, index, length	< 1	glib/glist.c Doubly-Linked list 170 LOC	free, prepend, reverse, nth, nth_data, position, find, index, last, length	< 1
	remove_all	4	OpenBSD Queue 70 LOC	simpleq_init, simpleq_insert_head, simpleq_insert_tail, simpleq_remove_head	< 1
	copy	5			
	merge_sorted_lists	98			
	insert_sorted_list	42			
	merge_sort	20			
SV-COMP Heap Manipulation 150 LOC	alloc_or_die_slave, dll_insert_slave, dll_create_slave, dll_destroy_slave, list_head_init, list_head_add, list_del	< 1	ExpressOS Memory Region 80 LOC	memory_region_init, create_user_space_region, split_memory_region	< 1
GRASS-hopper [31] Singly-Linked List 160 LOC	sl_concat, sl_copy, sl_dispose, sl_insert, sl_reverse, sl_traverse1, sl_traverse2	< 1	GRASS-hopper [31] Singly-Linked List 130 LOC	rec_concat, rec_copy, rec_dispose, rec_filter, rec_insert, rec_remove, rec_reverse, rec_traverse	< 1
	sl_filter, sl_remove	3			
GRASS-hopper [31] Doubly-Linked List 170 LOC	dl_concat, dl_copy, dl_dispose, dl_insert, dl_remove, dl_reverse, dl_traverse	< 1	GRASS-hopper [31] Sorted List I 80 LOC	sls_concat, sls_dispose, sls_reverse, sls_traverse1, sls_traverse2, merge_sort_rec	< 1
	dl_filter	5			
GRASS-hopper [31] Sorted List II 270 LOC	merge_sort_split	3	AFWP [20] Singly-Linked and Doubly-Linked List 240 LOC	SLL_create, SLL_delete_all, SLL_delete, SLL_filter, SLL_find, SLL_last, SLL_merge, SLL_reverse, SLL_rotate, SLL_swap, DLL_fix, DLL_splice	< 1
	sls_pairwise_sum	3			
	sls_insert	3			
	sls_remove	1			
	sls_filter	2			
	insertion_sort	30			
	sls_merge	7			
	sls_double_all	39		SLL_insert	3
	sls_copy	55			

Table 1: Experimental results of verification of 152 routines

mula abstraction to uninterpreted functions. Chin et al. [7] also exploit recursion, but reduce verification to standard logical theories that are not necessarily decidable.

Comparison with other deductive verification tools: There are, in general, three layers of annotations common in current deductive verification tools: (A) the specification, written as pre/post/assert annotations in code, (B) loop invariants and strengthening of pre/-post conditions that encode *inductive* invariants, and (C) proof tactic advises from the user to the underlying prover to prove the resulting verification conditions.

Annotations of type (A) that encode specifications are always required. Most current tools, like VeriFast [30], Bedrock [8], VCC [9], and DAFNY [24], require all three levels (A, B, and C) of annotations for proving code correct.

The tool VCDRYAD presented in this paper and our previous tool in [32] requires A and B annotations only, relieving the programmer from writing annotations of type C, for data-structure verification.

The annotations encoding proof tactics (C) are clearly the hardest to write for a programmer, and require typically expertise in the workings of the underlying prover. We now describe what these proof-tactic annotations look like in the other tools, to give an idea of the relief of burden we provide by our work to the programmer.

VCDRYAD vs VCC: We first compare our tool with VCC, over which we build. VCC compiles to the lower-level logical language BOOGIE, which generates verification conditions and passes them to an SMT solver. The SMT solvers are given formulas in undecidable theories but are aided by triggers using E-matching, model-quantifier instantiation, etc. [17], and these tactics can be specified at the level of the C program itself using VCC.

Consider an implementation of a function `list_find` that finds a key in singly-linked, and returns 1 iff the key is found [5]. Note that VCC does not support separation logic, and hence even the specification is complex and very hard to write (ghost variables are used to represent the set of keys in the list, and to map values to nodes). Invariants specify that abstract keys in the list correspond to data fields of nodes owned by the list and that pointers from the list nodes point to the nodes in the list structure. Moreover, acyclicity is encoded by assigning a strictly increasing ghost number to each node in the list and guarantee that each node can be reached by following the list.

Also, VCC requires hints using assertions (see lines 33 and 41), without which it would not be able to those hints VCC would not be able to prove the loop invariants. In more complex examples, VCC would even need explicit triggers [6].

VCDRYAD allows separation logic specfications which are considerably more succinct and morever, does not need any additional guidance in terms of proof tactics.

VeriFast, Bedrock: The VeriFast tool for C/Java works with separation logic specifications, requires proof guidance at the code-level from the user. The system itself aims to do very little *search*, and hence gives predictably high performance. Consider an implementation of binary search tree insertion (`bst_insert`) in VeriFast [7] and VCDRYAD, with specifications in separation logic. Verification of `bst_insert` method in VeriFast (lines 105–146) relies on user's help in form of annotations at particular program locations to unbundle (`open`) heaplets (lines 109, 127, 130) and re-bundle (`close`) heaplets (117, 119, 121, 128, 131, 136, 142), as well as three user-provided lemmas. Moreover, the user also needs to manually instantiate `tree_add_inorder` lemma at lines 135 and 141. The verification of the bst insert implementation in VCDRYAD (28–61) requires no annotations for helping proofs.

The tool Bedrock [8] provides Coq libraries and allows users to give proof tactics at the code-level, and automates significant parts of the reasoning. Consider an implementation of an in-place singly linked list reversal in Bedrock (from its tutorial)[8] and VCDRYAD. In this example, Bedrock and VCDRYAD versions do not prove exactly the same property because Bedrock uses Coq's list type family to abstractly represent memory location while VCDRYAD reasons about list structure and set of keys stored in the list. However, Bedrock does require hints to be provided by the user; user-provided administrative lemmas for each separation logic predicate (lines 9–13), to relate memory representation to abstract representation using four more lemmas that are need to be packaged together using a Bedrock tactic (lines 15–44). No such user intervention is needed to prove C version using VCDRYAD.

Finally, jStar [15] is a tool for Java against separation logic specifications that uses abstract shape analysis for inferring invariants and using a theorem prover similar to Smallfoot. In this context, we can see our work on VCDRYAD as essentially providing a deep embedding of separation logic into tools that use a classical logic pipeline, where arbitrary nested quantification is avoided using a restricted separated logic and verification is automated using natural proofs.

Comparison with our earlier work [32]: We are crucially building on the work reported in [32]. The difference between work reported here and that in [32] is that the latter is for a toy language while the current work is for the C language. The C language for has features such as (weakly enforced) types, base types (including `char`, `unsigned`, `long`), casting, aggregate data-types using `struct`, pointers to a structure inside `struct`, operators on bits, low-level access to memory, etc. The toy language does not contain these features. In particular, the toy language had only *one* type of structure (location) with a *fixed* set of field pointers for it. This means that we cannot model two different structures in this language or structures containing pointers to the other structures.

For instance, in our benchmark suite, we verified a process address space manipulation in a secure OS (ExpressOS), using a data structure `MemReg`. This is a nested `struct`, and has a pointer to `Backingfile struct`. While we prove this C program correct, the work in [32] cannot handle it, and in fact the corresponding version in [32] simplifies the program so that it does not involve nested `struct` constructs, by removing the inner `struct BackingFile`.

Furthermore, our work here giving natural proofs for C over the tool framework VCC gives a general platform for verification, where programmers can get some proofs (involving data-structures) automated using natural proofs but are still allowed to use VCC for proving more complex specifications using the automatically proved properties.

Comparison with Liquid Types: At a very high level, we believe that natural proofs and *liquid types* [22, 34] are related and are in a sense *dual* to each other: while we take a logical deductive verification formalism and search intentionally for simple proofs, in liquid types, the type-checking mechanism is simple but types are enriched using general logic formulas. Despite the similarities in aiming for simple sound but incomplete proofs for properties expressed in logic, the approaches are mechanically radically different, and a formal comparison seems hard. The work on liquid types also has built-in invariant generation (using a form of the Houdini algorithm [16]) that we do not have yet for natural proofs. On the other hand, our work augments a verification framework where users can interact further to prove their programs correct using the powerful mechanisms VCC provides, while such an integration of liquid types into a deductive verifier seems harder.

7. Conclusions and Future Work

We have built a powerful technique and tool, by adapting the ideas of natural proofs for separation logic [32], to automate verification proofs of data-structure manipulation in a widely used systems programming language. The primary technical contribution is to show how natural proofs, though defined only at the verification-

[5] `http://www.cs.illinois.edu/~madhu/vcdryad/cmp/VCDryad_vs_VCC.html`

[6] `http://vcc.codeplex.com/SourceControl/latest#vcc/Test/testsuite/vacid-0/RedBlackTrees.c`

[7] `http://www.cs.illinois.edu/~madhu/vcdryad/cmp/VCDryad_vs_Verifast.html`

[8] `http://www.cs.illinois.edu/~madhu/vcdryad/cmp/VCDryad_vs_Bedrock.html`

condition level, can be encoded at the code-level, utilizing existing frameworks that handle the language semantics and memory model. The resulting tool VCDRYAD gives a powerful extension of VCC for sequential programs, building automaticity for the most difficult task of verifying the dynamic data-structures a program maintains.

Natural proofs, augmented with axioms for the data-structures that relate different recursive definitions, have been able to verify all data-structure examples we have tried. Typically, for a data-structure, once we have related the various recursive definitions using axioms, programs manipulating the data-structures seldom require more help.

However, natural proofs currently do not work for data-structures that cannot be defined recursively (such us DAGs, graphs, etc.). We have been able to prove some properties of Schorr-Waite algorithm only for trees, but not for general graphs.

Several future directions are interesting. First, it would be interesting to see how VCDRYAD can be used for verifying larger pieces of code, and how the programmer's manual interactions for proving more complex properties can be orchestrated with the automatically generated annotations provided by VCDRYAD. Second, while the current work has focused on automatic proof tactics, loop invariants (and strengthening pre/post conditions so that they become inductive) is a hard task for the programmer, and automating this, especially for DRYAD specifications, would advance the usability of deductive verification tools further.

Acknowledgments

We thank our shepherd, Swarat Chaudhuri, and the anonymous reviewers for their feedback in improving the paper. This work was partially supported by NSF CAREER award #0747041 and NSF Expeditions in Computing project ExCAPE (awards #1138994 and #1139056).

References

[1] 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*, pages 364–387, 2006.

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

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, pages 52–68, 2005.

[4] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, pages 115–137, 2006.

[5] D. Beyer. Competition on software verification (sv-comp), 2013. URL `http://sv-comp.sosy-lab.org/2014/`.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI'06*, 2006.

[7] 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. *Sci. Comput. Program.*, pages 1006–1036, 2012.

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

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

[10] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for C. *Electron. Notes Theor. Comput. Sci.*, pages 85–103, 2009.

[11] E. Cohen, W. J. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *SOFSEM'13*, pages 1–27, 2013.

[12] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *POPL'09*, pages 302–314, 2009.

[13] B. Cook, C. Haase, J. Ouaknine, M. J. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR'11*, pages 235–249, 2011.

[14] L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD'09*, pages 45–52, 2009.

[15] D. Distefano and M. J. Parkinson J. jStar: Towards practical verification for Java. In *OOPSLA '08*, pages 213–226, 2008.

[16] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME '01*, pages 500–517, 2001.

[17] Y. Ge and L. M. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV'09*, pages 306–320, 2009.

[18] C. Haase, S. Ishtiaq, J. Ouaknine, and M. J. Parkinson. SeLoger: A tool for graph-based reasoning in separation logic. In *CAV'13*, pages 790–795, 2013.

[19] R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE-24*, pages 21–38, 2013.

[20] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV*, pages 756–772, 2013.

[21] 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*, pages 41–55, 2011.

[22] M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI'09*, pages 304–315, 2009.

[23] C. Le Goues, K. R. M. Leino, and M. Moskal. The Boogie verification debugger. In *SEFM'11*, pages 407–414, 2011.

[24] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, pages 348–370, 2010.

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

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

[27] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI'01*, June 2001.

[28] J. A. Navarro Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI'11*, PLDI '11, pages 556–566, 2011.

[29] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, pages 1–19, 2001.

[30] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Sci. Comput. Program.*, 82:77–97, 2014.

[31] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *CAV'13*, 2013.

[32] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI '13*, pages 231–242, 2013.

[33] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[34] P. M. Rondon. *Liquid Types*. PhD thesis, UCSD, 2012.

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

[36] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS'11*, pages 298–315, 2011.

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