Predictable Verification using Intrinsic Definitions

ANONYMOUS AUTHOR(S)

We propose a novel mechanism of defining data structures using *intrinsic definitions* that avoids recursion and instead utilizes *monadic maps satisfying local conditions*. We show that intrinsic definitions are a powerful mechanism that can capture a variety of data structures naturally. We show that they also enable a predictable verification methodology that allows engineers to write ghost code to update monadic maps and perform verification using reduction to decidable logics. We evaluate our methodology using Boogie and prove a suite of data structure manipulating programs correct.

1 INTRODUCTION

In computer science in general, and program verification in particular, classes of finite structures (such as data structures) are commonly defined using *recursive definitions (aka inductive definitions)*. Proving that a set of structures is in such a class or proving that structures in the class have a property is naturally performed using *induction*, typically mirroring the recursive structure in its definition. For example, trees in pointer-based heaps can be defined using the following recursive definition in first-order logic (FOL) with least fixpoint semantics for definitions:

$$\begin{aligned} tree(x) &::=_{lfp} x = nil \lor \big(x \neq nil \land tree(l(x)) \land tree(r(x)) \\ &\qquad \land x \notin htree(l(x)) \land x \notin htree(r(x)) \land htree(l(x)) \cap htree(r(x)) = \emptyset\big) \\ htree(x) &::=_{lfp} ite\,(x = nil,\ \emptyset,\ htree\,(l(x)) \cup htree\,(r(x)) \cup \{x\}) \end{aligned} \tag{1}$$

In the above, *htree* maps each location x in the heap to the set of all locations reachable from x using l and r pointers, and the definition of *tree* uses this to ensure that the left and right trees are disjoint from each other and the root. Definitions in separation logic are similar (with heaplets being implicitly defined, and disjointness expressed using the separating conjunction '$\star$' [46, 47, 54]).

When performing imperative program verification, we annotate programs with loop invariants and contracts for methods, and reduce verification to validation of Hoare triples of the form $\{\alpha\}s\{\beta\}$, where s is a straight-line program (potentially with calls to other methods encoded using their contracts). The validity of each Hoare triple is translated to a pure logical validity question, called the *verification condition* (VC). When α and β refer to data structure properties, the resulting VCs are typically proved using induction on the structure of the recursive definitions. Automation of program verification reduces to automating validity of the logic the VCs are expressed in.

Logics that are powerful enough to express rich properties of data structures are invariably incomplete, not just undecidable, i.e., they do not admit any automated procedure that is complete (guaranteed to eventually prove any valid theorem, but need not terminate on invalid theorems). For instance, validity is incomplete for both first-order logic with least fixpoints and separation logic. Consequently, though verification frameworks like Dafny [34] support rich specification languages, validation of verification conditions can fail even for valid Hoare triples. Automated verification engines hence support several heuristics resulting in sound but incomplete verification.

When proofs succeed in such systems, the verification engineer is happy that automation has taken the proof through. However, when proofs *fail*, as they often do, the verification engineer is stuck and perplexed. First, they would crosscheck to see whether their annotations are strong enough and that the Hoare triples are indeed valid. If they believe they are, they do not have clear guidelines to help the tool overcome its incompleteness. Engineers are instead required to know the *underlying proof mechanisms/heuristics* the verification system uses in order to figure out why the system is unable to succeed, and figure out how to help the system. For instance, for data structures with recursive definitions, the proof system may just unfold definitions a few times, and the engineer must be able to see why this heuristic will not be able to prove the theorem and formulate new inductively provable lemmas or quantification triggers that can help. Such *unpredictable* verification systems that require engineers to know their internal heuristics and proof mechanisms frustrate verification experience.

Predictable Verification. In this paper, we seek an entirely new paradigm of *predictable* verification. We want a technique where:

(a) the verification engineer is asked to provide upfront a set of annotations that help prove programs correct, where these annotations are entirely *independent* of the verification mechanisms/tools, and

(b) the program verification problem, given these annotations, is guaranteed to be *decidable* (and preferably decidable using efficient engines such as SMT solvers).

The upfront agreement on the information that the verification engineer is required to provide makes their task crystal clear. The fact that the verification is decidable given these annotations ensures that the verification engine, given enough resources of time and space (of course) will eventually return proving the program correct or showing that the program or annotations are incorrect. There is no second-guessing by the engineer as the verification will never fail on valid theorems, and hence they need not worry about knowing how the verification engine works, or give further help. Note that the verification *without annotations* can (and typically will be) undecidable.

Intrinsic Definitions of Data Structures. In this paper, we propose an entirely new way of defining data structures, called *intrinsic definitions*, that facilitates a predictable verification paradigm for proving their maintenance. Rather than defining data structures using recursion, like in equation (1) above (which naturally calls for inductive proofs and invariably entails incompleteness), we define data structures by augmenting each location with additional information using *ghost maps* and demanding that certain *local conditions* hold between each location and its neighbors.

Intrinsic definitions formally require a set of monadic maps (maps of arity one) that associate values to each location in a structure (we can think of these as ghost fields associated with each location/object). We demand that the monadic maps on local neighborhoods of every location satisfy certain logical conditions. The existence of maps that satisfy the local logical conditions ensures that the structure is a valid data structure.

For example, we can capture trees in pointer-based heaps in the following way. Let us introduce maps $tree : Loc \to Bool$, $rank : Loc \to \mathbb{Q}^+$ (non-negative rationals), and $p : Loc \to Loc$ (for "parent"), and demand the following local property:

$$\begin{aligned}\forall x :: Loc.(tree(x) \Rightarrow (\ & (l(x) \neq nil \Rightarrow (tree(l(x)) \land p(l(x)) = x \land rank(l(x)) < rank(x)))\\ & \land (r(x) \neq nil \Rightarrow (tree(r(x)) \land p(r(x)) = x \land rank(r(x)) < rank(x)))\\ & \land ((l(x) \neq nil \land r(x) \neq nil) \Rightarrow l(x) \neq r(x))\\ & \land (p(x) \neq nil \Rightarrow (r(p(x)) = x \lor l(p(x)) = x)))\ )\end{aligned}$$

The above demands that ranks become smaller as one descends the tree, that a node is the parent of its children, and that a node is either the left or right child of its parent.

Given a *finite* heap, it is easy to see that if there exist maps *tree*, *rank* and p that satisfy the above property, and if $tree(l)$ is true for a location l, then l must point to a tree (strictly decreasing ranks ensure that there are no cycles and existence of a unique parent ensures that there are no "merges"). Furthermore, in any heap, if T is the subset of locations that are roots of trees, then there are maps that satisfy the above property and have precisely $tree(l)$ to be true for locations in T.

Note that the above intrinsic definition *does not use recursion* or least fixpoint semantics. It simply requires maps such that each location satisfies the local neighborhood condition.

Fix-what-you-break program verification methodology.

Intrinsic definitions are particularly attractive for proving *maintenance* of structures when structures undergo mutation. When a program mutates a heap H to a heap H', we start with monadic maps that satisfy local conditions in the pre-state. As the heap H is modified, we ask the verification engineer to also *repair* the monadic maps, using ghost map updates, so that the local conditions on all locations are met in the heap in the post-state H'.

For instance, consider a program that walks down a tree from its root to a node x and introduces a newly allocated node n between x and x's right child r. Then we would assume in the precondition that the monadic maps *tree*, *rank*, and p exist satisfying the local condition (2) above. After the mutation, we would simply update these maps so that $tree(n)$ is true, $p(r) = n$, $p(n) = x$, and $rank(n)$ is, say, $(rank(x) + rank(r))/2$.

The annotations required of the user, therefore, are ghost map updates to locations such that the local conditions are valid for each location. We will guarantee that checking whether the local conditions holds for each location, after the repairs, is expressible in decidable logics.

We propose a modular verification approach for verifying data structure maintenance that asks the programmer to fix what they break. Given a program that we want to verify, we instead verify an *augmented program* that keeps track of a ghost set of *broken locations Br*. Broken locations are those that (potentially) do not satisfy the local condition. When the program destructively modifies the fields of an object/location, it and some of its neighbors (accessible using pointers from the object) may not satisfy the local condition anymore, and hence will get added to the broken set. The verification engineer must repair the monadic maps on these broken locations and ensure (through an assertion) that the local condition holds on them before removing them from the broken set Br. However, even while repairing monadic maps on a location, the local condition on *its neighboring* locations may fail and get added to the broken set.

We develop a *fix-what-you-break (FWYB)* program verification paradigm, giving formal rules of how to augment programs with broken sets, how users can modify monadic maps, and fixed recipes of how broken sets are maintained in any program. In order to verify that a method m maintains a data structure, we need to prove that if m starts with the broken set being empty, it returns with the empty broken set. We prove this methodology sound, i.e., if the program augmented with broken sets and ghost updates is correct, then the original program maintains the data structure properties mentioned in its contracts.

Decidable Verification of Annotated Programs. The general idea of using local conditions to capture global properties has been explored in the literature to reduce the complexity of proofs (e.g., iterated separation in separation logic [55]; see Section 6). Intrinsic definitions of data structures and the fix-what-you-break program verification methodology are more specifically designed to ensure the key property of *decidable verification of annotated programs* by avoiding both recursion/least-fixpoint definitions and avoiding even quantified reasoning.

The verification conditions for Hoare triples involving basic blocks of our annotated programs have the following structure. First, the precondition can be captured using *uninterpreted monadic functions* that are *implicitly* assumed to satisfy the local condition on each location that is not in the broken set Br (avoiding universal quantification). The monadic map updates (repairs) that the verification engineer makes can be captured using map updates. The postcondition of the ghost-code augmented program can, in addition to properties of variables, assert properties of the broken set Br using logics over sets. Finally, we show that capturing the modified heap after function calls can be captured using *parameterized map update* theories, that are decidable [18]. Consequently, the entire verification condition is captured in quantifier-free logics involving maps, parametric map updates, and sets over combined theories. These verification conditions are hence decidable and efficiently handled by modern SMT solvers[1].

Intrinsic Definitions for Representative Data Structures and Verification in Boogie. Intrinsic definitions of data structures is a novel paradigm and capturing data structures requires thinking anew in order to formulate monadic maps and local conditions that characterize them.

We give intrinsic definitions for several classic data structures such as linked lists, sorted lists, circular lists, trees, binary search trees, AVL trees, and red-black trees. These require novel definitions of monadic maps and local conditions. We also show how standard methods on these data structures (insertions, deletions, concatenations, rotations, balancing, etc.) can be verified using the fix-what-you-break strategy and standard loop invariant/contract annotations. We also consider *overlaid data structures* consisting of multiple data structures overlapping and sharing locations. In particular, we model the core of an overlaid data structure that is used in an I/O scheduler in Linux that has a linked list (modeling a FIFO queue) overlaid on a binary search tree (for efficient search over a key field). Intrinsic definitions beautifully capture such structures by compositionally combining the instrinsic definitions for each structure and a local condition linking them together. We show methods to modify this structure are provable using fix-what-you-break verification.

We model the above data structures and the annotated methods in the low-level programming language Boogie. Boogie in an intermediate programming language with verification support that several high-level programming languages compile to for verification Boogie (e.g., C [15, 16], Dafny [34], Civl [27], Move [19]). These annotated programs do not use quantifiers or recursive definitions, and Boogie is able to verify them automatically using decidable verification in negligible time, without further user-help.

Contributions. The paper makes the following contributions:

- A new paradigm of *predictable verification* that asks upfront for programmatic annotations and ensures annotated program verification is decidable, without reliance on users to give heuristics and tactics.
- A novel notion of intrinsic definitions of data structures based on ghost monadic maps and local conditions.
- A predictable verification methodology for programs that manipulate data structures with intrinsic definitions following a fix-what-you-break (FWYB) methodology.
- Intrinsic definitions for several classic data structures, and fix-what-you-break annotations for programs that manipulate such structures, with realization of these programs and their verification using Boogie.

2 INTRINSIC DEFINITIONS OF DATA STRUCTURES: THE FRAMEWORK

In this section we present the first main contribution of our paper, the framework of intrinsically defined data structures. We first define the notion of a data structure in a pointer-based heap.

2.1 Data Structures

In this paper, we think of data structures defined using a *class* C of objects. The class C can coexist with other classes, heaps, and data structures, potentially modeled and reasoned with using other mechanisms. For technical exposition and simplicity, we restrict the technical definitions to a single class of data structures over a class C.

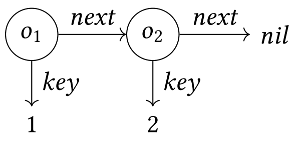
A class C has a signature $(\mathcal{S}, \mathcal{F})$ consisting of a finite set of sorts $\mathcal{S} = \{\sigma_0, \sigma_1 \dots, \sigma_n\}$ and a finite set of fields $\mathcal{F} = \{f_1, f_2 \dots, f_m\}$. We assume without loss of generality that the sort σ_0 represents

[1]Assuming of course that the underlying quantifier-free theories are decidable; for example, integer multiplication in the program or in local conditions would make verification undecidable, of course.

the sort of objects of the class C, and we denote this sort by C itself. We use C to model objects in the heap. The other "background" sorts, e.g., integers, are used to model the values of the objects' fields. Each field $f_i : C \longrightarrow \sigma$ is a unary function symbol and is used to model pointer and data fields of heap locations/objects. We model *nil* as a non-object value and denote the sort $C \uplus \{nil\}$ consisting of objects as well as the *nil* value by $C?$.

A C-heap H is a *finite* first-order model of the signature C. More formally, it is a pair (O, I) where O is a finite set of *objects* interpreting the foreground sort C and I is an interpretation of every field in $\mathcal{F}$ for every object in O.

Example 2.1 (C-Heap). Let C be the class consisting of a pointer field $next : C \longrightarrow C?$ and a data field $key : C \longrightarrow Int$. The figure on the right represents a C-heap consisting of objects $O = \{o_1, o_2\}$ and the illustrated interpretation I for *next* and *key*. □



We now define a data structure. We fix a class C.

Definition 2.2 (Data Structure). A data structure D of arity k is a set of triples of the form $(O, I, \overline{o})$ such that (O, I) is a C-heap and $\overline{o}$ is a k-tuple of objects from O. □

Informally, a data structure is a particular subset of C-heaps along with a distinguished tuple of locations $\overline{o}$ in the heap that serve as the "entry points" into the data structure, such as the root of a tree or the ends of a linked list segment.

Example 2.3 (Sorted Linked List). Let C be the class defined in Example 2.1. The data structure of sorted linked lists is the set of all (O, I, o_1) such that O contains objects $o_1, o_2 \ldots o_n$ with the interpretation $next(o_i) = o_{i+1}$ and $key(o_i) \leq key(o_{i+1})$ for every $1 \leq i < n$, and $next(o_n) = nil$. For example, let (O, I) be the C-heap described in Example 2.1. The triple (O, I, o_1) is an example of a sorted linked list. Here o_1 represents the head of the sorted linked list. □

2.2 Intrinsic Definitions of Data Structures

In this work, we propose a characterization of data structures using *intrinsic definitions*. Intrinsic definitions consist of a set of *monadic maps* that associate (ghost) values to each object and a set of *local* conditions that constrain the monadic maps on each location and its neighbors. A C-heap is considered to be a valid data structure if *there exists* a set of monadic maps for the heap that satisfy the local conditions.

Annotations using intrinsic definitions enable local and decidable reasoning for correctness of programs manipulating data structures using the Fix-What-You-Break (FWYB) methodology, which is described later in Section 3. We develop the core idea of intrinsic definitions below.

Ghost Monadic Maps. We denote by $C_{\mathcal{G}} = (\mathcal{S}, \mathcal{F} \cup \mathcal{G})$ an extension of C with a finite set of monadic (i.e., unary) function symbols $\mathcal{G}$. We can think of these as *ghost* fields of objects.

The key idea behind intrinsic definitions is to extend a C-heap with a set of ghost monadic maps and formulate local conditions using the maps that characterize the heaps belonging to the data structure. The existence of such ghost maps satisfying the local conditions is then the intrinsic definition. Definitions are parameterized by a multi-sorted first-order logic $\mathcal{L}$ in which local conditions are stated. The logic has the sorts $\mathcal{S}$ and contains the function symbols in $\mathcal{F} \cup \mathcal{G}$, as well as interpreted functions over background sorts (such as $+$ and $<$ on integers, and $\subseteq$ on sets).

Definition 2.4 (Intrinsic Definition). Let $C = (\mathcal{S}, \mathcal{F})$ be a class. An intrinsic definition $IDS(\overline{y})$ over the class C is a tuple $(\mathcal{G}, \mathcal{L}, LC, \varphi(\overline{y}))$ where:

(1) $\mathcal{G}$ is a finite set of *monadic map names and signatures* disjoint from $\mathcal{F}$,

(2) $\mathcal{L}$ is a first-order logic over the sorts $\mathcal{S}$ containing the interpreted functions of the background sorts as well as the function symbols in $\mathcal{F} \cup \mathcal{G}$,
(3) A *local condition* formula LC of the form $\forall x : Loc.\, \rho(x)$ such that ρ is a quantifier-free $\mathcal{L}$-formula, and
(4) A *correlation formula* $\varphi(\overline{y})$ that is a quantifier-free $\mathcal{L}$-formula over free variables $\overline{y} \in Loc$. □

We denote an intrinsic definition by $(\mathcal{G}, LC, \varphi(\overline{y}))$ when the logic $\mathcal{L}$ is clear from context. In this work $\mathcal{L}$ is typically a decidable combination of quantifier-free theories [43, 44, 58], containing theories of integers, sets, arrays [18], etc., supported effectively in practice by SMT solvers [7, 17].

Definition 2.5 (Data Structures defined by Intrinsic Definitions). Let $C = (\mathcal{S}, \mathcal{F})$ be a class and $IDS(\overline{y}) = (\mathcal{G}, LC, \varphi(\overline{y}))$ be an intrinsic definition over C consisting of monadic maps $\mathcal{G}$, local condition LC and correlation formula φ. The data structure defined by IDS is precisely the set of all $(O, I, \overline{o})$ where *there exists* an interpretation J that extends I with interpretations for the symbols in $\mathcal{G}$ such that $O, J \models LC$ and $O, J[\overline{y} \mapsto \overline{o}] \models \varphi(\overline{y})$, where $[\overline{y} \mapsto \overline{o}]$ denotes that the free variables $\overline{y}$ are interpreted as $\overline{o}$.

Informally, given a data structure DS consisting of triples $(O, I, \overline{o})$, an intrinsic definition demands that there exist monadic maps $\mathcal{G}$ such that the C-heaps (O, I) in the data structure can be extended with values for maps in $\mathcal{G}$ satisfying the local conditions LC, and the entrypoints $\overline{o}$ are characterized in the extension by the quantifier-free formula φ.

Example 2.6 (Sorted Linked List). Recall the data structure of sorted linked lists defined in Example 2.3. We capture sorted linked lists by an intrinsic definition $SortedLL(y)$ using monadic maps $sortedll : C \longrightarrow Bool$ and $rank : C \longrightarrow \mathbb{Q}^{+}$ such that:

$$LC \equiv \forall x. \Big((sortedll(x) \;\wedge\; next(x) \neq nil) \Rightarrow \\ (sortedll(next(x)) \;\wedge\; rank(next(x)) < rank(x) \;\wedge\; key(x) \leq key(next(x))) \Big)$$
$$\varphi(y) \equiv \; sortedll(y)$$

In the above definition the $rank$ field decreases wherever $sortedll$ holds as we take the $next$ pointer, and hence assures that there are no cycles. Observe that without the constraint on $rank$, the triple $(\{o_1, o_2\}, I, o_1)$ where $I = \{next(o_1) = o_2, next(o_2) = o_1, key(o_1) = key(o_2) = 0\}$ denoting a two-element circular list would satisfy the definition, which is undesirable.

Note that the above allows for a heap to contain both sorted lists as well as unsorted lists. We are guaranteed by the local condition that the set of all objects where $sortedll$ is true will be the heads of sorted lists.

We can also replace the domain of ranks in the above definition using any strict partial order, say integers or reals (with the usual $<$ order on them), and the definition will continue to define sorted lists. Well-foundedness of the order is not important as heaps are *finite* in our work (see definition of C-heaps in Section 2.1) □

3 FIX WHAT YOU BREAK (FWYB) VERIFICATION METHODOLOGY

In this section we present the second main contribution of this paper: the Fix-What-You-Break (FWYB) methodology. We begin by describing a while programming language and defining the verification problem we study. We fix a class $C = (\mathcal{S}, \mathcal{F})$ throughout this section.

3.1 Programs, Contracts, and Correctness

Programs. Figure 1 shows the programming language used in this work. Note that we can use variables and expressions over non-object sorts. Functions can return multiple outputs. We assume

$$\begin{aligned} P :=\ & x := nil \mid x := y \mid v := be \mid y := x.f \mid v := x.d \\ & \mid x.f := y \mid x.d := v \mid x := \text{new } C() \mid \overline{r} := Function(\overline{t}) \\ & \mid \text{skip} \mid \text{assume } cond \mid \text{return} \mid P\,;P \mid \text{if } cond \text{ then } P \text{ else } P \mid \text{while cond do } P \\ cond :=\ & x = y \mid x \neq y \mid be \quad \text{(Condition Expressions)} \end{aligned}$$

Fig. 1. Grammar of while programs with recursion. x, y are variables denoting objects of class C? (i.e., C objects or *nil*), v, w are a background sort(s) variables, r, t denote variables of any sort, f is a pointer field, d is a data field, and be is a expression of the background sort(s).

that method signatures contain designated output variables and therefore the return statement does not mention values.

Our language is safe (i.e., allocated locations cannot point to un-allocated locations) and garbage-collected. Formally we consider configurations θ consisting of a store (map from variables to values) and a heap along with an error state $\perp$ to model error on a null dereference. We denote that a formula α is satisfied on a configuration θ by writing $\theta \models \alpha$.

Intrinsic Hoare Triples. The verification problem we study in this paper is *maintenance* of data structure properties. Fix an intrinsic definition $(\mathcal{G}, LC, \varphi(\overline{y}))$ where $\mathcal{G} = \{g_1, g_2 \ldots, g_k\}$. Let $\overline{z}$ be the input/output variables for a program that we want to verify. We consider pre and post conditions of the form

$$\exists\, g_1, g_2 \ldots, g_k.\ (LC \wedge \varphi(\overline{w}) \wedge \psi(\overline{z}))$$

where each g_i is a ghost monadic map (unary function over locations), ψ is a quantifier-free formula over $\overline{z}$ that can use the ghost monadic maps g_i, and $\overline{w}$ is a tuple of variables from $\overline{z}$ whose arity is equal to $\overline{y}$. Note that the above has a second-order existential quantification ($\exists$) over function symbols $g_1, \ldots, g_k$, and LC has first-order universal quantification over a single location variable. Read in plain English, "$\overline{w}$ points to a data structure *IDS* such that the (quantifier-free) property $\psi(\overline{z})$ holds".

We study the validity of the following Hoare Triples:

$$\langle\, \alpha(\overline{x})\,\rangle\ \ \mathrm{P}(\overline{x},\ ret\colon\ \overline{r})\ \ \langle\, \beta(\overline{x}, \overline{r})\,\rangle$$

where α and β are pre and post conditions of the above form, P is a program, and $\overline{x}, \overline{r}$ are input and output variables for P respectively.

Example 3.1 (Running Example: Insertion into a Sorted List). Let $SortedLL(y) = (\mathcal{G}, LC, sorted(y))$ as in Example 2.6 where $\mathcal{G} = \{sortedll, rank\}$. The following Hoare triple says that insertion into a sorted list returns a sorted list:

$$\langle\, \exists\, sortedll, rank.\ LC \wedge sortedll(x)\,\rangle\ sorted\text{–}insert(x, k,\ ret\colon x)\ \langle\, \exists\, sortedll, rank.\ LC \wedge sortedll(x)\,\rangle$$

where x, r are variables of type C, k is of type *Int* and *sorted–insert* is the usual recursive method.

Validity of Intrinsic Hoare Triples. We now define the validity of Hoare Triples.

Definition 3.2 (Validity of Intrinsic Hoare Triples). An intrinsic triple $\langle\, \alpha\,\rangle\, P\, \langle\, \beta\,\rangle$ is *valid* if for every configuration θ such that $\theta \models \alpha$, transitioning according to P starting from θ does not encounter the error state $\perp$, and furthermore, if θ transitions to θ' under P, then $\theta' \models \beta$.

An Overview of FWYB

We develop the Fix-What-You-Break (FWYB) methodology in three stages, in the following subsections. We give here an overview of the methodology and the stages.

Recall that intrinsic triples are of the form $\langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \alpha)\, \rangle\; P\; \langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \beta)\, \rangle$. In Stage 1 (Section 3.2) we remove the second-order quantification. We do this by requiring the verification engineer to explicitly construct the g_i maps in the post state from the maps in the pre state using *ghost code*. We then obtain triples of the form $\langle\, LC \wedge \varphi \wedge \alpha\, \rangle\; P_{\mathcal{G}}\; \langle\, LC \wedge \varphi \wedge \beta\, \rangle$ where $P_{\mathcal{G}}$ is an augmentation of P with ghost code that updates the $\mathcal{G}$ maps.

Note that the LC in the contract universally quantifies over objects. In Stages 2 (Section 3.3) and 3 (Section 3.4) we remove the quantification by explicitly tracking the objects where the local conditions do not hold and treating them as implicitly true on all other objects. We call this set Br the *broken set*. Intuitively, the broken set grows when the program mutates pointers or makes other changes to the heap, and shrinks when the verification engineer repairs the $\mathcal{G}$ maps using ghost code to satisfy the LC on the broken objects. The specifications assume an empty broken set at the beginning of the program and the engineer must ensure that it is empty again at the end of the program. However, they do not have to track the objects manually. We develop in Stage 3 (Section 3.4) a discipline for writing only *well-behaved* manipulations of the broken set. This reduces the problem to triples of the form $\langle\, \varphi \wedge \alpha\, \rangle\; P_{\mathcal{G},Br}\; \langle\, \varphi \wedge \beta\, \rangle$, where $P_{\mathcal{G},Br}$ contains ghost code for updating both $\mathcal{G}$ and Br. Note that these specifications are quantifier-free, and checking them can be effectively automated using SMT solvers [7, 17].

3.2 Stage 1: Removing Existential Quantification over Monadic Maps using Ghost Code

Consider an intrinsic Hoare Triple $\langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \alpha)\, \rangle\; P\; \langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \beta)\, \rangle$. Read simply, the precondition says that *there exist* maps $\{g_i\}$ satisfying some properties, and the postcondition says that we must *show the existence* of maps $\{g_i\}$ satisfying the post state properties.

We remove existential quantification from the problem by re-formulating it as follows: we assume that we are *given* the maps $\{g_i\}$ as part of the pre state such that they satisfy $LC \wedge \varphi \wedge \alpha$, and we require the verification engineer to *compute* the $\{g_i\}$ maps in the post state satisfying $LC \wedge \varphi \wedge \beta$. The engineer computes the post state maps by taking the given pre state maps and 'repairing' them on an object whenever the program breaks local conditions on that object. The repairs are done using *ghost code*, which is a common technique in verification literature [23, 26, 36, 53].

Aside: Ghost Code. We do not formalize ghost code here as it is standard; please see prior literature for a formal exposition [23, 26, 36, 53]. Intuitively, ghost variables/fields cannot influence the computation of non-ghost variables/fields. In particular, this means that when conditional statements or loops use ghost variables in the condition, the body of the statement must also consist entirely of ghost code. These conditions can be checked statically. We must also ensure that such 'ghost loops' are always terminating since a nonterminating ghost loop changes the meaning of the original program. We do this by requiring the verification engineer to write a ranking function that decreases with each iteration. We treat ghost recursive functions similarly. We henceforth assume the validity of a Hoare Triple for program containing ghost loops/recursive functions to include the condition that the given ranking function must decrease. We also use below the idea of 'projecting out' the ghost code, which refers to the original program with all ghost code simply eliminated. Note that the definition of ghost code ensures that the projection operation is sensible.

Formally, fix an intrinsically defined data structure $(\mathcal{G}, LC, \varphi)$. We extend the class signature $C = (\mathcal{S}, \mathcal{F})$ (and consequently the programming language) to $C_{\mathcal{G}} = (\mathcal{S}, \mathcal{F} \cup \mathcal{G})$ and treat the symbols in $\mathcal{G}$ as *ghost fields* of objects of class C in the program semantics. Performing the transformation described above reduces the verification problem to proving triples of the form $\langle\, LC \wedge \varphi \wedge \alpha\, \rangle\; P_{\mathcal{G}}\; \langle\, LC \wedge \varphi \wedge \beta\, \rangle$, where there is no existential quantification over $\mathcal{G}$ and $P_{\mathcal{G}}$ is an augmentation of P with ghost code that updates the $\mathcal{G}$ maps. The following proposition captures the correctness of this reduction:

Proposition 3.3. *Let ψ_{pre} and ψ_{post} be quantifier-free formulae over $\mathcal{F} \cup \mathcal{G}$. If $\langle\, \psi_{pre} \,\rangle\ P_{\mathcal{G}}\ \langle\, \psi_{post} \,\rangle$ is valid then $\langle\, \exists g_1, g_2 \ldots, g_k.\, \psi_{pre} \,\rangle\ P\ \langle\, \exists g_1, g_2 \ldots, g_k.\, \psi_{post} \,\rangle$ is valid* [2]*, where P is the projection of $P_{\mathcal{G}}$ obtained by eliminating ghost code.*

We note a point of subtlety here: the obtained triple eliminates existential quantification over $\mathcal{G}$ by claiming something stronger than the original specification, namely that for *any* maps $\{g_i\}$ such that ψ_{pre} is satisfied in the pre state, there is a *computation* that yields corresponding maps in the post state such that ψ_{post} holds. The onus of coming up with such a computation is placed on the verification engineer.

3.3 Stage 2: Relaxing Universal Quantification using Broken Sets

We turn to verifying programs whose pre and post conditions are of the form $LC \wedge \gamma$, where $LC \equiv \forall z.\, \rho(z)$ is the local condition. Consider a program P that maintains the data structure. The local conditions are satisfied everywhere in both the pre and post state of P. However, they need not hold everywhere in the intermediate states. In particular, P may call a method N which may neither receive nor return a proper data structure. To reason about P modularly we must be able to express contracts for methods like N. To do this we must be able to talk about program states where only some objects may satisfy the local conditions.

Broken Sets. We introduce in programs a ghost set variable Br that represents the set of (potentially) broken objects. Intuitively, at any point in the program the local conditions must always be satisfied on every object that is *not* in the broken set. Formally, for a program P we extend the signature of P with Br as an additional input and an additional output. We also write pre and post conditions of the form $(\forall z \notin Br.\, \rho(z)) \wedge \gamma$ to denote that local conditions are satisfied everywhere outside the broken set, where γ can now use Br. In particular, given the Hoare triple

$$\langle\, (\forall z.\, \rho) \wedge \alpha \,\rangle\ P_{\mathcal{G}}(\overline{x},\ ret\colon\ \overline{y})\ \langle\, (\forall z.\, \rho) \wedge \beta \,\rangle$$

from Stage 1, we instead prove the following Hoare triple (whose validity implies the validity of the triple above):

$$\langle\, (\forall z \notin Br.\, \rho) \wedge \alpha \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho) \wedge \beta \wedge Br = \emptyset \,\rangle$$

where Br is a ghost input variable of the type of set of objects and $P_{\mathcal{G},Br}$ is an augmentation of P with ghost code that computes the $\mathcal{G}$ maps as well as the Br set satisfying the postcondition.

P may also call other methods N with bodies Q. We similarly extend the input and output signatures of the called methods and use the broken set to write appropriate contracts for the methods, introducing triples of the form $\langle\, (\forall z \notin Br.\, \rho) \wedge \alpha_N \,\rangle\ Q_{Br}(\overline{s}, Br,\ ret:\ \overline{r}, Br)\ \langle\, (\forall z \notin Br.\, \rho) \wedge \beta_N \,\rangle$. Again, $Q_{\mathcal{G},Br}$ is an augmentation of Q with ghost code that updates $\mathcal{G}$ and Br.

For the main method that preserves the data structure property, the broken set is empty at the beginning and end of the program. However, called methods or loop invariants can talk about states with nonempty broken sets. We require the verification engineer to write ghost code that maintains the broken set accurately. The correctness argument for this stage is similar to Proposition 3.3.

3.4 Stage 3: Eliminating the Universal Quantifier for Well-Behaved Programs

We consider triples of the form

$$\langle\, (\forall z \notin Br.\, \rho) \wedge \alpha \,\rangle\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \langle\, (\forall z \notin Br.\, \rho) \wedge \beta \,\rangle$$

where $P_{\mathcal{G},Br}$ is a program augmented with ghost updates to the $\mathcal{G}$-fields as well as the Br set, and α, β are quantifier-free formulae that can also mention the fields in $\mathcal{G}$ and the Br set. In this stage

[2]Here the notion validity for both triples is given by Definition 3.2, where configurations are interpreted appropriately with or without the ghost fields.

we would like to eliminate the quantified conjunct entirely and instead ask the engineer to prove the validity of the triple

$$\{\alpha\}\ P_{\mathcal{G},Br}(\overline{x}, Br,\ ret\colon\ \overline{y}, Br)\ \{\beta\}$$

However, the above two triples are not, in general, equivalent (as broken sets can be manipulated wildly). In this section we define a syntactic class of *well-behaved* programs that force the verification engineer to maintain broken sets correctly, and for such programs the above triple are indeed equivalent. For example, for a field mutation, well-behaved programs require the engineer to determine the set of *impacted objects* where local conditions may be broken by the mutation. The well-behavedness paradigm then mandates that the engineer add the set of impacted objects to the broken set immediately following the mutation statement. Similarly, well-behaved programs do not allow the engineer to remove an object from the broken set unless they show that the local conditions hold on that object. The imposition of this discipline ensures that programmers carefully preserve the meaning of the broken set (i.e., objects outside the broken set must satisfy local conditions). This allows for the quantified conjunct in the triple obtained from Stage 2 to be dropped since it always holds for a well-behaved program. Let us look at such a program:

Example 3.4 (Well-Behaved Sorted List Insertion). We use the running example (Example 3.1) of insertion into a sorted list. We consider a snippet where where the key k to be inserted lies between the keys of x and $next(x)$ (which we assume is not nil). We ignore the conditionals that determine $next(x) \neq nil$ and $key(x) \leq k \leq key(next(x))$ for brevity.

We first relax the universal quantification as described in Stage 2 (Section 3.3) and rewrite the pre and post conditions to $(\forall\, z \notin Br.\, LC(z)) \wedge sortedll(x) \wedge Br = \emptyset$. Making the first conjunct implicit, we write the following program that manipulates the broken set in a well-behaved manner. We show the value of the broken set through the program in comments on the right:

```
pre: sortedll(x) ∧ Br = ∅
post: sortedll(x) ∧ Br = ∅
 assert x ∉ Br;
 assume LC(x);
 y := x.next;     // {}
 z := new C();
 Br := Br ∪ {z}; // {z}
 z.key := k;
 Br := Br ∪ {z}; // {z}
 z.next := y;
 Br := Br ∪ {z}; // {z}
z.sortedll := True;
Br := Br ∪ {z}; // {z}
x.next := z;
Br := Br ∪ {x}; // {x,z}
z.rank := (x.rank + y.rank)/2;
Br := Br ∪ {z}; // {x,z}
// x and z satisfy LC
assert LC(z);
Br := Br \ {z}; // {x}
assert LC(x);
Br := Br \ {x}; // {}
```

We depict the statements enforced by the well-behavedness paradigm in pink and the ghost updates written by the verification engineer in blue. Observe that the paradigm adds the impacted objects to the broken set after each mutation and allocation. Determining the impact set of a mutation is nontrivial; we show how to construct them in Section 4.1. Note also that to remove x from the broken set we must show $LC(x)$ holds (assert followed by removal from Br). Finally, we see at the beginning of the snippet that if we show $x \notin Br$ then we can infer that $LC(x)$ holds. This follows from the meaning of the broken set.

Putting it All Together. The above program corresponds to the program $P_{\mathcal{G},Br}$ obtained from the Stage 3 reduction, consisting of ghost updates to the $\mathcal{G}$ maps and Br. Since it is well-behaved and satisfies the contract $\langle\, sortedll(x) \wedge Br = \emptyset\,\rangle\ P_{\mathcal{G},Br}\ \langle\, sortedll(x) \wedge Br = \emptyset\,\rangle$ we can conclude that it satisfies the contract $\langle\, (\forall z \notin Br.\, \rho) \wedge sortedll(x) \wedge Br = \emptyset\,\rangle\ P_{\mathcal{G},Br}\ \langle\, (\forall z \notin Br.\, \rho) \wedge sortedll(x) \wedge Br = \emptyset\,\rangle$. Using Proposition 3.3 we can project out the ghost code and conclude that the triple given in Example 3.1 with the user's original program and intrinsic specifications is valid! In this way, using

FWYB we can verify programs with respect to intrinsic specifications by verifying augmented programs with respect to quantifier-free specifications. The latter can be discharged efficiently in practice using SMT solvers [7, 17]. □

Aside: Generating Quantifier-Free Verification Conditions. We argue above that verifying augmented programs with quantifier-free specifications reduces to validity over combinations of quantifier-free theories. However, unlike scalar programs, quantifier-free contracts do not guarantee quantifier-free verification conditions (VCs) for heap programs. In particular, commands such as allocation and function calls pose challenges. However, we show that in our case it is indeed possible to obtain quantifier-free VCs. We do this by transforming a given heap program into a scalar program that explicitly models changes to the heap. We model allocation using a ghost set *Alloc* corresponding to the allocated objects and update it when a new object is allocated. We reason about arbitrary changes to the heap across a function call by requiring a 'modifies' annotation from the user and adding assumptions that the fields of objects outside the modified set of a function call remain the same across the call. We detail this reduction in Appendix A.1.

We dedicate the rest of this section to developing the general theory of well-behaved programs.

Rules for Constructing Well-Behaved Programs

We define the class of well-behaved programs using a set of rules. We first introduce some notation.

We distinguish the triples over the augmented programs and quantifier-free annotations by $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, with $\{\}$ brackets rather than $\langle\rangle$. We denote that a triple is provable by $\vdash \{\psi_{pre}\}\ P\ \{\psi_{post}\}$. Our theory is agnostic to the underlying mechanism for proving triples correct (we use the off-the-shelf verification tool Boogie in our evaluation). However, we assume that the mechanism is sound with respect to the operational semantics. We denote that a snippet P is well-behaved by $\vdash_{\mathrm{WB}} P$. We also denote that local conditions hold on an object x by $LC(x)$.

Figure 2 shows the rules for writing well-behaved programs. We only explain the interesting cases here.

Mutation. Since mutations can break local conditions, we must grow the broken set. Let A be a finite set of object-type terms over x such that for any $z \notin A$, if $LC(z)$ held before the mutation, then it continues to hold after the mutation. We refer to such a set A as an *impact set* for the mutation, and we update Br after a mutation with its impact set. The impact set may not always be expressible as a finite set of terms, but this is indeed the case for all the intrinsically defined data structures we use in this paper. We show how to construct impact sets in Section 4.1.

Allocation. Allocation does not modify the heap on any existing object. Therefore, we simply update the broken set by adding the newly created object x (this was also the case in Example 3.4).

Assert LC and Remove. This rule allows us to shrink the broken set once the verification engineer fixes the local conditions on a broken location. The snippet `assert LC(x); Br := Br\{x}` in Example 3.4 uses this rule. Informally, the verification engineer is required to show that $LC(x)$ holds before removing x from Br.

Infer LC Outside Br. Recall that for well-behaved programs we know implicitly that $\forall x \notin Br.\ \rho(x)$ holds. This rule allows us to instantiate this implicit fact on objects that we can show lie outside the broken set. The snippet in `assert x∉Br; assume LC(x)` Example 3.4 uses this rule.

In the above presentation we use only one broken set for simplicity of exposition. Our general framework allows for finer-grained broken sets that can track breaks over a partition on the local conditions. For example, in Section 4.4 we verify deletion in an overlaid data structure consisting of a linked list and a binary search tree using two broken sets: one each for the local conditions of the two component data structures.

Skip/Assignment/Lookup/Return

$$\frac{}{\vdash_{\mathrm{WB}}\ s \text{ where } s \text{ is of the form skip, x:=y, x:=y.f, or return}}$$

Mutation

$$\frac{\vdash \{\, z \notin A \wedge LC(z) \wedge x \neq nil \,\}\ x.f := v\ \{\, LC(z) \,\}}{\vdash_{\mathrm{WB}}\ x.f := v\,;\ Br := Br \cup A \quad \text{where } A \text{ is a finite set of location terms over } x}$$

Allocation

$$\frac{}{\vdash_{\mathrm{WB}}\ x := \mathrm{new}\, C()\,;\ Br := Br \cup \{x\}}$$

Function Call

$$\frac{}{\vdash_{\mathrm{WB}}\ \overline{y}, Br := Function(\overline{x}, Br)}$$

Infer LC Outside Br

$$\frac{}{\vdash_{\mathrm{WB}}\ \text{if } (x \neq nil \wedge x \notin Br) \text{ then assume } LC(x)}$$

Assert LC and Remove

$$\frac{}{\vdash_{\mathrm{WB}}\ \text{if } LC(x) \text{ then } Br := Br \setminus \{x\}}$$

Composition

$$\frac{\vdash_{\mathrm{WB}}\ P \qquad \vdash_{\mathrm{WB}}\ Q}{\vdash_{\mathrm{WB}}\ P\,;\,Q}$$

If-Then-Else

$$\frac{\vdash_{\mathrm{WB}}\ P \qquad \vdash_{\mathrm{WB}}\ Q}{\vdash_{\mathrm{WB}}\ \text{if } cond\ P \text{ else } Q \quad \text{where } cond \text{ does not mention } Br}$$

While

$$\frac{\vdash_{\mathrm{WB}}\ P}{\vdash_{\mathrm{WB}}\ \text{while } cond \text{ do } P \quad \text{where } cond \text{ does not mention } Br}$$

Fig. 2. Rules for constructing well-behaved programs. Local condition formula instantiated at x is denoted by $LC(x)$. The statement (if *cond* then S) is sugar for (if *cond* then S else skip).

3.5 Soundness of FWYB

In this section we state the soundness of the FWYB methodology. We first define some terminology.

Fix a main method M with input variables $\overline{x}, Br$ and output variables $\overline{y}, Br$. Let the body of M be P. Let $N_i, 1 \leq i \leq k$ be a set of auxiliary methods that P calls with bodies Q_i, where the methods N_i also have similar signatures with Br as the last input parameter and last output parameter. Note that the bodies P and Q_i contain updates of ghost fields. Let us denote by a program the collection of methods $[(M : P); (N_1 : Q_1) \ldots (N_k : Q_k)]$. We define the projection of M to a user-level program:

Definition 3.5 (Projection of Augmented Code to User Code). The projection of the augmented program $[(M : P); (N_1 : Q_1) \ldots (N_k : Q_k)]$ is the user-level program $[(M' : P'); (N'_1 : Q'_1) \ldots (N'_k : Q'_k)]$ such that:

(1) If M has signature $(\overline{x}, Br,\ ret\colon \overline{y}, Br)$, then M' has signature $(\overline{x},\ ret\colon \overline{y})$. Similarly for each $N_i, 1 \leq i \leq k$, the corresponding method N'_i removes Br from the signature.

(2) P' is derived from P by: (a) eliminating all ghost code, and (b) replacing each function call statement of the form $\overline{r}, Br := N_j(\overline{z}, Br)$ with the statement $\overline{r} := N_j(\overline{z})$. Similarly each Q'_i is derived from the corresponding Q_i by a similar transformation.

We now state the soundness theorem.

Theorem 3.6 (FWYB Soundness). *Let $(\mathcal{G}, LC, \varphi)$ be an intrinsic definition with $\mathcal{G} = \{g_1, g_2 \ldots, g_l\}$. Let $[(M : P); (N_1 : Q_1) \ldots, (N_k : Q_k)]$ be an augmented program constructed using the FWYB methodology such that $\vdash_{\mathrm{WB}} P$ and $\vdash_{\mathrm{WB}} Q_i$, $1 \leq i \leq k$, i.e., the programs P and Q_i are well-behaved (according to the rules in Figure 2). Let ψ_{pre} and ψ_{post} be quantifier-free formulae that do not mention Br (but can mention the maps in $\mathcal{G}$). Finally, let $[(M' : P'); (N'_1 : Q'_1) \ldots, (N'_k : Q'_k)]$ be the projected user-level program according to Definition 3.5. Then, if the triple:*

$$\{\varphi \wedge \psi_{pre} \wedge Br = \emptyset\}\ P\ \{\varphi \wedge \psi_{post} \wedge Br = \emptyset\}$$

is valid, then the triple

$$\langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \wedge \varphi \wedge \psi_{pre})\,\rangle\; P'\; \langle\, \exists\, g_1, g_2 \ldots, g_l.\, (LC \wedge \varphi \wedge \psi_{post})\,\rangle$$

is valid (according to Definition 3.2). □

Informally, the soundness theorem says that given a user-written program, if we augment it with updates to ghost fields and the broken set only using the discipline for well-behaved programs and show that if the broken set is empty at the beginning of the program it will be empty at the end, then the original user-written program, with contracts on preservation of the data structure, is correct. We provide a proof of the theorem in Appendix B.

4 ILLUSTRATIVE DATA STRUCTURES AND VERIFICATION

Intrinsic definitions and the fix-what-you-break verification methodology are new concepts that require thinking afresh about data structures and annotating methods that operate over them. In this section, we present several classical data structures and methods over them, and illustrate how the verification engineer can write intrinsic definitions (which maps to choose, and what the local conditions ensure) and how they can fix broken sets to prove programs correct.

4.1 Insertion into a Sorted List

In this section we present the verification of insertion into a sorted list implemented in the FWYB methodology in its entirety. Our running example in Section 3 illustrates the key technical ideas involved in verifying the program. In this section we present an end-to-end picture that mirrors the verification experience in practice.

Data Structure Definition. We first revise the definition of a sorted list (Example 2.6) with a different set of monadic maps. We have the following monadic maps $\mathcal{G}$— $prev : C \longrightarrow C?$, $length : C \longrightarrow \mathbb{N}$, $keys : C \longrightarrow Set(Int)$, $hslist : C \longrightarrow Set(C)$ that model the *previous* pointer (inverse of next), length of the sorted list, the set of keys stored in it, and its heaplet (set of locations that form the sorted list) respectively. We use the length, keys, and heaplet maps to state full functional specifications of methods. The local conditions are:

$$\begin{aligned}
&\forall x.\ next(x) \neq nil \Rightarrow (\ key(x) \leq key(next(x)) \ \wedge\ prev(next(x)) = x \\
&\qquad\qquad\qquad\qquad \wedge\ length(x) = 1 + length(next(x)) \ \wedge\ keys(x) = \{key(x)\} \cup keys(next(x)) \\
&\qquad\qquad\qquad\qquad \wedge\ hslist(x) = \{x\} \uplus hslist(next(x))\ ) \qquad\qquad (\uplus\text{: disjoint union}) \\
&\ \wedge\ prev(x) \neq nil \Rightarrow next(prev(x)) = x \\
&\ \wedge\ next(x) = nil \Rightarrow (\ length(x) = 1 \ \wedge\ keys(x) = \{key(x)\} \ \wedge\ hslist(x) = \{x\}\ )
\end{aligned} \tag{2}$$

The above definition is slightly different from the one given in Example 2.6. The *length* map replaces the *rank* map, requiring additionally that lengths of adjacent nodes differ by 1.

The *prev* map is a gadget we find useful in many intrinsic definitions. The constraints on *prev* ensure that the C-heaps satisfying the definition only contain non-merging lists. To see why this is the case, consider for the sake of contradiction distinct objects o_1, o_2, o_3 such that $next(o_1) = next(o_2) = o_3$. Then, we can see from the local conditions that we must simultaneously have $prev(o_3) = o_1$ and $prev(o_3) = o_2$, which is impossible. Finally, the *hslist* and *keys* maps represent the heaplet and the set of keys stored in the sorted list (respectively).

The heads of all sorted lists in the C-heap is then defined by the following correlation formula:

$$\varphi(y) \equiv prev(y) = nil$$

Constructing Provably Correct Impact Sets for Mutations. We now instantiate the rules developed in Section 3.4 for sorted lists. Recall that well-behaved programs must update the broken

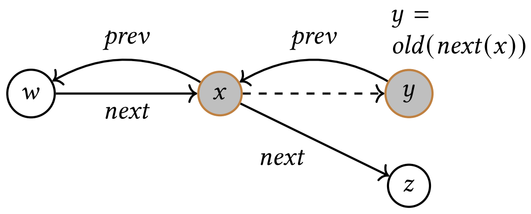


Fig. 3. Reasoning about the set of objects broken by `x.next := z`. The dashed arrow represents the old *next* pointer before the mutation. The grey nodes denote objects where local conditions can be broken by the mutation. We see that only x and y may violate *next* and *prev* being inverses.

Mutated Field f	Impacted Objects A_f
$x.next$	$\{x, old(next(x))\}$
$x.key$	$\{x, prev(x)\}$
$x.prev$	$\{x, old(prev(x))\}$
$x.hslist$	$\{x, prev(x)\}$
$x.length$	$\{x, prev(x)\}$
$x.keys$	$\{x, prev(x)\}$

Table 1. Table of impact sets corresponding to field mutations for sorted lists (See 2 in Section 4.1). $old(t)$ refers to the value of the term t before the mutation. Terms only belong to the sets if not equal to *nil*.

set with the impact set of a mutation. Table 1 captures the impact set for each field mutation. Note that the terms denoting the impacted objects belong to A_f only if they do not evaluate to *nil*.

Let us consider the correctness of Table 1, focusing on the mutation of *next* as an example. Figure 3 illustrates the heap after the mutation `x.next := z`. We make the following key observation: the local constraints $LC(v)$ for an object v refer only to the properties of objects v, $next(v)$, and $prev(v)$ (see 2), i.e., objects that are at most "one step" away on the heap. Therefore, the only objects that can be broken by the mutation `x.next := z` are those that are one step away from x either via an incoming or an outgoing edge via pointers *next* and *prev*. This is a general property of intrinsic definitions: *mutations cannot immediately affect objects that are far away on the heap.* [3]

In our case, we claim that the impact set contains at most x and $old(next(x))$. Here's a proof (see Fig 3): Consider z such that $z \neq old(next(x))$ (as there is no real mutation otherwise). If z was not broken before the mutation, then it cannot be the case that $prev(z) = x$. Looking at the local conditions, it is clear that such a z will remain unbroken after the mutation. Now consider a w not broken before the mutation such that $next(w) = x$. Then it follows from the local conditions that there can only be one such (unbroken) w, and further $w \neq x$. w's fields are not mutated, and by examining LC, it is easy to see that w will not get broken (as $LC(v)$ does not refer to $next(next(v))$). The argument is the same for w such that $prev(x) = w$. Finally, consider a y not broken before the mutation such that $prev(y) = x$. We can then see from the local conditions that $y = old(next(x))$, which is already in the impact set.

The above argument is subtle, but we can easily automatically check whether impact sets declared by a verification engineer are correct. The MUTATION rule in Figure 2 characterizes the impact set A_f for mutation of a field f thus:

$$\vdash \{u \neq x \land u \neq next(x) \land LC(u) \land x \neq nil\}\ x.next := z\ \{LC(u)\}$$

The above says that any location u that is not in the impact set that satisfied the local conditions must continue to satisfy them after the mutation. Note that the validity of this triple is decidable. In our realization of the FWYB methodology we prove our broken sets correct by encoding the triple in BOOGIE (see Section 5.3).

Macros that ensure Well-Behaved Programs. In Section 3.4 we characterized well-behaved programs as a set of syntactic rules (Figure 2). We can realize these restrictions using macros:

[3]Note that a mutation can necessitate changes to monadic maps for an unbounded number of nodes *eventually*; however, these are not necessary immediately. As we fix monadic maps on a broken object, its neighbors could get broken and need to be fixed, leading to their neighbors breaking, etc. This can lead to a ripple effect that would eventually require an unbounded number of locations to be fixed.

(1) `Mut(x,f,v,Br)` for each $f \in \mathcal{F} \cup \mathcal{G}$, which represents the sequence of statements `x.f := v; Br := Br U` $\mathtt{A_f(x)}$. Here $\mathtt{A_f(x)}$ is the impact set corresponding to the mutation on f on x as given by the table above. This macro is used instead of `x.f := v` and automatically ensures that the impact set is added to the broken set.
(2) `NewObj(x,Br)`, which represents the statements `x := new C(); Br := Br U { x }`. This macro is used instead of `x := new C()` and ensures that any newly allocated object is automatically added to the broken set.
(3) `AssertLCAndRemove(x,Br)`, which represents the statements `assert LC(x); Br := Br \ { x }`. This macro is allowed anytime the engineer wants to assert that x satisfies the local condition, and then remove it from the broken set.[4]
(4) `InferLCOutsideBr(x, Br)`, which represents the statements `assert (x ≠ nil ∧ x ∉ Br); assume LC(x)`. This allows the engineer at any time to assert that x is not in the broken set and assume it satisfies the local condition.

The above macros correspond to the rules Mutation, Allocation, Assert LC and Remove, and Infer LC Outside Br respectively. Restricting to the syntactic fragment that contains the above macros and disallows mutation and allocation otherwise enforces the *programming discipline* that ensures well-behaved programs.

We present the full well-behaved code written using the above macros and discuss it in Appendix C.1.

4.2 Reversing a Sorted List

We return to lists for another case study: reversing a sorted list. The purpose of this example is to demonstrate how the fix-what-you-break philosophy works with iteration/loops. We augment the definition of sorted linked lists from Case Study 4.1 to make sortedness optional and determined by predicates that capture sortedness in non-descending order, with $sorted : C \longrightarrow Bool$, and sortedness with non-ascending order, with $rev\_sorted : C \longrightarrow Bool$. The relevant additions to the local condition and the impact sets for these monadic maps can be seen below:

$$\begin{aligned}
&(next(x) \neq nil \Rightarrow \\
&\quad sorted(x) \Rightarrow key(x) \leq key(next(x)) \wedge sorted(x) = sorted(next(x)) \\
&\qquad \wedge\ rev\_sorted(x) \Rightarrow key(x) \geq key(next(x)) \\
&\qquad \wedge\ rev\_sorted(x) = rev\_sorted(next(x)))
\end{aligned}$$

Mutated Field f	Impacted Objects A_f
$sorted$	$\{x, prev(x)\}$
rev_sorted	$\{x, prev(x)\}$

We present the full local condition and code in Appendix C.3. However, the gist of the method is that we are popping C nodes off of the front of a temporary list cur, and pushing them to the front of a new reversed list ret. The method consists mainly of a loop which performs the aforementioned action repeatedly. A technique we use to verify loops using FWYB is to maintain that the broken set contains no nodes or only a finite number of nodes for which we specify how they are broken. In the case of this method, Br remains empty, as the loop maintains cur and ret as two valid lists, not modifying any other nodes. When popping x from cur and adding it to ret, in addition to repairing the new cur by setting its parent pointer to nil, we also need to update fields such as $length$ and $keys$ on x, so it satisfies the relevant local conditions as the new head of the ret list. We also need to set the predicate rev_sorted to be true on x, as ret is sorted in the opposite direction to cur.

4.3 Circular Lists

Our next example is circular lists. This example illustrates a neat trick in FWYB that where we assert that we can reach a special node known as a *scaffolding* node, and that in addition to asserting

[4]We extend our basic programming language defined in Figure 1 with an assert statement and give it the usual semantics (program reaches an error state if the assertion is not satisfied, but is equivalent to skip otherwise).

properties on the node that is given to the method, one can also assert properties on this scaffolding node. In order to make verification of properties on this scaffolding node easier, the scaffolding node remains unchanged in the data structure, and is never deleted. We start with a data structure containing a pointer $next : C \longrightarrow C$ and a monadic map $prev : C \longrightarrow C$. We build on this data structure to define circular lists by adding a monadic map $last : C \longrightarrow C$ where $last(u)$ for any location u points to the last item in the list, which ends up being the scaffolding node in this case. The last item in the list, x must in turn point to another node whose *last* map points to x itself: this ensures cyclicity. We also define monadic maps $length : C \longrightarrow Nat$ and $rev\_length : C \longrightarrow Nat$ to denote the distance to the *last* node by following *prev* or *next* pointers. The partial local conditions for x are as below:

$$\begin{gathered}(x = last(x) \Rightarrow last(next(x)) = x \wedge length(x) = 0 \wedge rev\_length(x) = 0)\\ \wedge\ (x \neq last(x) \Rightarrow last(next(x)) = last(x) \wedge length(x) = length(next(x)) + 1\\ \wedge\ rev\_length(x) = rev\_length(prev(x)) + 1)\end{gathered}$$

Here is the gist of inserting a node at the back of a circular list. We are given a node x such that $next(x) = last(x)$ (at the end of a cycle). We insert a newly allocated node after x, making local repairs there. Then, in a ghost loop similar to the one in Case Study 4.2, we make appropriate updates to the *length* and *keys* maps, which are not fully described here, following the *prev* map until we reach $last(x)$.

4.4 Overlaid Data Structure of List and BST

One of the settings where intrinsic definitions shine is in defining and manipulating an *overlaid data structure* that overlays a linked list and a binary search tree. The list and tree share the same locations, and the *next* pointer threads them into a linked list while the *left, right* pointers on them defines a BST. Such structures are often used in systems code (such as Linux kernels) to save space [33]. For example, I/O schedulers use an overlaid structure as above, where the list/queue stores requests in FIFO order while the bst enables faster searching according requests with respect to a key. While there has been work in verification of memory safety of such structures [33], we aim here to check preservation of such data structures.

Intrinsic definition over such an overlaid data structure is pleasantly *compositional.* We simply take intrinsic definitions for lists and trees, and take the union of the monadic maps and the conjunction of their local conditions. The only thing that's left is then to ensure that they contain the same set of locations. We introduce a monadic map bst_{root} that maps every node to its root in the bst, and introduce a monadic map $list_{\text{head}}$ that maps every node to the head of the list it belong to (using appropriate local conditions). We then demand that all locations in a list have the same bst_{root} and all all locations in a tree have the same $list_{\text{head}}$, using local conditions. We also define monadic maps that define the bst-heaplet for tree nodes and list-heaplet for list nodes (the locations that belong to the tree under the node or the list from that node, respectively) using local conditions. We define a correlation predicate *Valid* that relates the head h of the list and root r of the tree by demanding that the bst-root of h is r and the list-head of the tree root is h, and furthermore, the list-heaplet of h and tree-heaplet of r are equal. This predicate can be seen here:

$$Valid \equiv\ bst\_root(h) = r \wedge list\_root(r) = h \wedge\ list\_heaplet(h) = bst\_heaplet(r)$$

We prove certain methods manipulating this overlaid structure correct (such as deleting the first element of the list and removing it both from the list as well as the BST). These ghost annotations are mostly compositional— we fix monadic maps for BST in the way we fix them for a stand-alone BST and fix monadic maps for lists in the same way we fix them for lists. In fact, we maintain two broken sets, one for BST and one for list, as updating a pointer for BST often doesn't break the local property for lists, and vice versa.

Limitations. In modeling the data structures above, we crucially used the fact that for any location, there is at most one location (or a bounded number of locations) that has a field pointing to this location. We used this fact to define an inverse pointer (*prev* or *parent*/*p*), which allows us to capture the impact set when a location's fields are mutated. Consequently, we do not know how to model structures where locations can have unbounded indegree. We could model these inverse pointers using a sequence/array of pointers, but verification may get more challenging. Data structures with unbounded outdegree can however be modeled using just a linked-list of pointers and hence seen as a structure with bounded outdegree.

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation Strategy of IDS and FWYB in Boogie

We implement the technique of intrinsically defined data structures and FWYB verification in the program verifier Boogie [6]. Boogie is a low-level imperative programming language which supports systematic generation of verification conditions that are checked using SMT solvers. We choose Boogie as it allows us a sufficient degree of control to ensure that the generated verification conditions are quantifier-free and fall into decidable logics. Furthermore, a plethora of higher-level languages compile to Boogie (e.g., VCC and Havoc for C [15, 16], Dafny [34] with compilation to .NET, Civl for concurrent programs [27], Move for smart contracts [19], etc.). Implementing a technique in Boogie hence shows a pathway for implementing IDS and FWYB for higher-level languages as well.

Modeling Fix-What-You-Break Verification in Boogie. We model heaps in Boogie by having a sort *Loc* of locations and modeling pointers as maps from *Loc* to sorts. We implement monadic maps also as maps from locations to field values. We implement our benchmarks using the *macros* for well-behaved programming defined in Section 4.1. We implement allocation with an *Alloc* set and heap change across function calls using parameterized map updates as described in Appendix A.1.

We ensure that the VCs generated by Boogie fall into decidable fragments, and there are several components that ensure this. First, note that all specifications (contracts and invariants) are quantifier-free. Second, pure functions (used to implement local conditions) are typically encoded using quantification, but we ensure Boogie inlines them to avoid quantification. Third, heap updates that are the effect of procedures and set operations for set-valued monadic maps are modeled using parameterized map updates [18]. Since Boogie does not natively support parameterized map updates, we define these map updates as uninterpreted functions and automatically augment the generated SMT file with concrete implementations based on map updates. In modeling all our benchmarks, we verify that the final SMT query is quantifier-free, albeit with parameterized map updates, which ensures that it is decidable.

5.2 Benchmarks

We evaluate our technique on a variety of data structures and methods that manipulate them. Our benchmark suite consists of data structure manipulation methods for a variety of different list and tree data structures, including sorted lists, circular lists, binary search trees, and balanced binary search trees such as Red-Black trees and AVL trees. Methods include core functionality such as search, insertion and deletion. The suite includes an *overlaid* data structure that overlays a binary search tree and a linked list, implementing methods needed by a simplified version of the Linux deadline IO scheduler [33]. The contracts for these functions are complete functional specifications that not only ask for maintenance of the data structure, but correctness properties involving the returned values, the keys stored in the container, and the heaplet of the data structure.

Data Structure	*LC* Size	Method	LOC+Spec +Ann	Verif. Time(s)	Method	LOC+Spec +Ann	Verif. Time(s)
Singly-Linked List	8	Append	4+11+10	2.5	Insert-Back	6+13+12	2.4
		Copy-All	7+8+9	2.5	Insert-Front	3+13+7	2.3
		Delete-All	10+9+16	2.5	Insert	9+13+23	2.4
		Find	4+4+2	2.2	Reverse	6+8+18	2.5
Sorted List	14	Delete-All	10+9+16	2.6	Merge	11+9+20	2.8
		Find	4+4+2	2.3	Reverse	5+14+22	2.7
		Insert	9+16+27	2.6			
Sorted List (w. *min*, *max* maps)	20	Concatenate	6+10+13	2.9	Find-Last	5+10+9	2.4
Circular List	27	Insert-Front	4+12+41	3.2	Delete-Front	3+12+39	3.2
		Insert-Back	5+14+45	3.3	Delete-Back	3+13+55	3.3
Binary Search Tree	35	Find	4+3+5	2.5	Delete	10+13+30	3.9
		Insert	9+12+37	3.5	Remove-Root	17+15+47	5.2
Treap	37	Find	4+3+5	2.5	Delete	10+13+30	4.0
		Insert	19+12+74	12.8	Remove-Root	24+15+74	6.9
AVL Tree	45	Insert	12+12+36	5.5	Find-Min	5+5+8	2.6
		Delete	43+13+62	6.5	Balance	40+17+95	6.4
Red-Black Tree	48	Insert	76+12+203	55.6	Del-L-Fixup	33+20+93	9.1
		Delete	56+13+76	7.2	Del-R-Fixup	33+20+93	9.0
		Find-Min	5+5+8	2.6			
BST+Scaffolding	59	Delete-Inside	1+24+51	5.9	Remove-Root	44+31+61	12.6
Scheduler Queue (overlaid SLL+BST)	72	Move-Request	4+10+8	4.6	BST-Delete-Inside	1+29+55	7.7
		List-Remove-First	5+13+10	3.9	BST-Remove-Root	44+36+65	18.9

Table 2. Implementation and verification of Boogie programs on the benchmarks. The columns give data structure, size of local conditions for capturing the datatructure as number of conjuncts, method, lines of executable code in the method, lines of specification (pre/post), lines of ghost code annotations (invariants/monadic map updates), and verification time in seconds.

5.3 Evaluation

We first evaluate the following two research questions:
RQ1: Can the data structures be expressed using IDS, and can the FWYB methodology for methods on these structures be expressed in Boogie?
RQ2: Is Boogie with decidable verification condition generation dispatched to SMT solvers effective in verifying these methods?

As we have articulated earlier, intrinsic definitions and monadic map updates require a new way of thinking about programs and repairs. We implement the specifications using monadic maps and local conditions, and the benchmarks using the well-behavedness macros and ghost updates. We were able to express all data structures and FWYB annotations for the methods on these structures for our benchmarks in Boogie (RQ1). Importantly, we were able to write quantifier-free modular contracts for the auxiliary methods and loop invariants using the monadic maps and strengthening the contracts using quantifier-free assertions on broken sets (which may not be empty for auxiliary methods). We do not prove termination for these methods except for ghost loops and ghost recursive procedures (termination for latter is required for soundness). We provide the benchmarks with annotations in an anonymized repository[5] and in the supplemental material.

Our annotation measures and verification results are detailed in the table in Table 2, for 42 methods across 10 data structure definitions. These measurements were taken from a machine with an Intel™ Core i5-4460 processor at 3.20 GHz. We found the verification performance excellent overall (RQ2): all the methods verify in under 1 minute, and all but four verify in under 10 seconds.

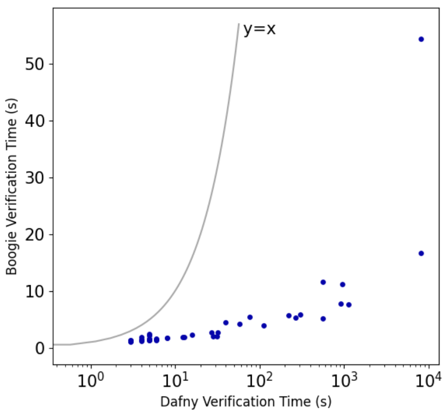
[5] https://zenodo.org/records/10146749

We used the option that sets the maximum number of VC splits to 8 in Boogie. The times reported for each method are the sum of times taken for the following steps: verifying that the impact sets are correct (<3s for all data structures), generating verification conditions with Boogie, injecting parametric update implementations, and solving the SMT queries.

Notice that the lines of ghost code written is nontrivial, but these are typically simple, involving programmatically repairing monadic maps and manipulating broken sets. In fact, a large fraction ($\sim$ 60%) of ghost updates in our benchmarks were *definitional updates* that simply update a field according to its definition in the local condition. An example is updating $x.length$ to $x.next.length+1$ for lists. We believe that the annotation burden can be significantly lowered in future work by automating such updates. More importantly, note that none of the programs required further annotations like instantiations, triggers, inductive lemmas, etc. in order to prove them correct.

RQ3: What is the performance impact of generating decidable verification conditions?

In order to study this, we implemented the entire benchmark suite in Dafny, a higher-level programming language that uses Boogie to perform its verification. We implemented the data structures and the FWYB methodology identically in Dafny as the Boogie version. Even though our annotations are all quantifier-free, Dafny generates Boogie code where several aspects of the language, in particular allocation and heap change across function calls, are modeled using *quantified* formulas, resulting in quantified queries to SMT solvers.



The scatter plot on the right shows the performance of Boogie and Dafny on the benchmarks. The plot clearly strongly suggests that even though Dafny is able to prove the FWYB-annotated programs correct, using decidable verification conditions results in much better performance. We hence believe that implementing program verifiers (such as Dafny) that exploit the fact that FWYB annotations can be compiled to annotations in Boogie that result in decidable VCs is a promising future direction to achieve faster high-level IDS+FWYB frameworks.

6 RELATED WORK

There have been mainly two paradigms to automated verification of programs annotated with rich contracts written in logic. The first is to restrict the specification logic so that verification conditions fall into a decidable logic. The second allows validity of verification conditions to fall into an undecidable or even an incomplete logic (where validity is not even recursively enumerable), but support effective strategies nevertheless, using heuristics, lemma synthesis, and further annotations from the programmer [2–5, 9, 10, 12–14, 20, 25, 42, 45, 46, 49, 51, 54, 57]. In this paper, we have proposed a new paradigm of predictable verification that calls for programmers to write a reasonable amount of extra annotations under which validity of verification conditions becomes decidable. To the best of our knowledge, we do not know of any other work of this style (where validity of verification conditions is undecidable but an upfront set of annotations renders it decidable).

Decidable verification. There is a rich body of research on decidable logics for heap verification: first-order logics with reachability [35], the logic Lisbq in the Havoc tool [31], several decidable fragments of separation logic known [8, 50] as well as fragments that admit a decidable entailment problem [22]. Decidable logics based on interpreting bounded treewidth data structures on trees have also been studied, for separation logics as well as other logics [24, 37, 38]. In general, these logics are heavily restricted— the magic wand in separation logic quickly leads to undecidability [11], the general entailment problem for separation logic with inductive predicates is undecidable [1], and validity of first-order logic with recursive definitions is undecidable and not even recursively enumerable and does not admit complete proof procedures.

Monadic Maps. Monadic maps have been exploited in earlier work in other forms for simplifying verification of properties of global structures. In shape analysis [56], monadic predicates are often used to express inductively defined properties of single locations on the heap. In separation logic, the *iterated separating conjunction operator*, introduced already by Reynolds in 2002 [55], expresses local properties of each location, and is akin to monadic maps. Iterated separation conjunction has been used in verification, for both arrays as well as for data structures, in various forms [21, 41]. The work on verification using flows [28–30, 39, 40, 48] introduces predicates based on flows, and utilizes such predicates in iterated separation formulas to express global properties of data structures and to verify algorithms such as the concurrent Harris list. In these works, local properties of locations and proof systems based on them are explored, but we do not know of any work exploiting monadic maps for decidable reasoning, which is crucial for predictable verification.

Ghost code. The methodology of writing ghost code is a common paradigm in deductive program verification [23, 26, 36, 53] and supported by verification tools such as Boogie and Dafny [6, 34]. Ghost code involves code that manipulates auxiliary variables to perform a parallel computation with the original code without affecting it. Our use of ghost code establishes the required monadic maps that satisfy local conditions by allowing the programmer to construct the maps and verify the local conditions using a disciplined programming methodology. Furthermore, we assure that the original code with the ghost code results in decidable verification problems, which is a salient feature not found typically in other contexts where ghost code is used.

7 CONCLUSIONS

We introduced intrinsic definitions that eschew recursion/induction and instead define data structures using monadic maps and local conditions. Proving that a program maintains a valid data structure hence requires only maintaining monadic maps and verifying the local conditions on locations that get broken. Furthermore, verifying that engineer-provided ghost code annotations are indeed correct falls into decidable theories, leading to a predictable verification framework.

Future Work. First, it would be useful to develop verification engines for higher-level languages (like Java [25], Rust [32], and Dafny [34]) that that have native support for intrinsic definitions and produce verification conditions in decidable theories that SMT solvers can handle (see RQ3 in Section 5.3). Second, it would be interesting to see how intrinsic definitions with fix-what-you-break proof methodology can coexist and exchange information with traditional recursive definitions with induction-based proof methodology. Third, as mentioned in Section 5.3, many updates of monadic maps are straightforward using definitions, and tools that automate this can reduce annotation burden significantly. Fourth, we are particularly intrigued with the ease with which intrinsic data structures capture more complex data structures such as overlaid data structures. Exploring intrinsic definitions for verifying concurrent and distributed programs that maintain data structures is particularly interesting. Finally, intrinsic definitions opens up an entirely new approach to defining properties of structures that simplify reasoning. We believe that exploiting intrinsic definitions in other verification contexts, like mathematical structures used in specifications (e.g., message queues in distributed programs), parameterized concurrent programs (configurations modeled as unbounded sequences of states), and programs that manipulate big data concurrently (like Apache Spark) are exciting future directions.

REFERENCES

[1] Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max Kanovich, and Joël Ouaknine. 2014. Foundations for Decision Problems in Separation Logic with General Inductive Predicates. In *Foundations of Software Science and Computation Structures*, Anca Muscholl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 411–425.

[2] Anindya Banerjee, Mike Barnett, and David A. Naumann. 2008. Boogie Meets Regions: A Verification Experience Report. In *Verified Software: Theories, Tools, Experiments*, Natarajan Shankar and Jim Woodcock (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 177–191.

[3] Anindya Banerjee and David A. Naumann. 2013. Local Reasoning for Global Invariants, Part II: Dynamic Boundaries. *J. ACM* 60, 3, Article 19 (jun 2013), 73 pages. https://doi.org/10.1145/2485981

[4] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2008. Regional Logic for Local Reasoning about Global Invariants. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 387–411.

[5] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2013. Local Reasoning for Global Invariants, Part I: Region Logic. *J. ACM* 60, 3, Article 18 (June 2013), 56 pages. http://doi.acm.org/10.1145/2485982

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 364–387.

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 171–177.

[8] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. A Decidable Fragment of Separation Logic. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, Kamal Lodaya and Meena Mahajan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–109.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *Programming Languages and Systems*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–68.

[10] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2006. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects* (Amsterdam, The Netherlands) *(FMCO'05)*. Springer-Verlag, Berlin, Heidelberg, 115–137. https://doi.org/10.1007/11804192_6

[11] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. 2008. On the Almighty Wand. In *Computer Science Logic*, Michael Kaminski and Simone Martini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 323–338.

[12] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (dec 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

[13] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2007. Automated Verification of Shape, Size and Bag Properties. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS '07)*. IEEE Computer Society, USA, 307–320. https://doi.org/10.1109/ICECCS.2007.17

[14] Duc-Hiep Chu, Joxan Jaffar, and Minh-Thai Trinh. 2015. Automatic Induction Proofs of Data-Structures in Imperative Programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 457–466. https://doi.org/10.1145/2737924.2737984

[15] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–42.

[16] Jeremy Condit, Brian Hackett, Shuvendu K. Lahiri, and Shaz Qadeer. 2009. Unifying Type Checking and Property Checking for Low-Level Code. *SIGPLAN Not.* 44, 1 (jan 2009), 302–314. https://doi.org/10.1145/1594834.1480921

[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[18] Leonardo de Moura and Nikolaj Bjørner. 2009. Generalized, efficient array decision procedures. In *2009 Formal Methods in Computer-Aided Design*. IEEE, 45–52. https://doi.org/10.1109/FMCAD.2009.5351142

[19] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. 2022. Fast and Reliable Formal Verification of Smart Contracts with the Move Prover. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 183–200.

[20] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, Holger Hermanns and Jens Palsberg (Eds.). Springer

Berlin Heidelberg, Berlin, Heidelberg, 287–302.

[21] Dino Distefano and Matthew Parkinson. 2008. jStar: Towards Practical Verification for Java. *Sigplan Notices - SIGPLAN* 43, 213–226. https://doi.org/10.1145/1449764.1449782

[22] Mnacho Echenim, Radu Iosif, and Nicolas Peltier. 2021. Unifying Decidable Entailments in Separation Logic with Inductive Definitions. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 183–199.

[23] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 2016. The spirit of ghost code. *Formal Methods in System Design* 48 (2016), 152–174.

[24] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. 2013. The Tree Width of Separation Logic with Recursive Definitions. In *Automated Deduction – CADE-24*, Maria Paola Bonacina (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–38.

[25] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 41–55.

[26] C. B. Jones. 2010. The Role of Auxiliary Variables in the Formal Development of Concurrent Programs. In *Reflections on the Work of C.A.R. Hoare*, A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood (Eds.). Springer London, London, 167–187. https://doi.org/10.1007/978-1-84882-912-1_8

[27] Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. 143–152. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_23

[28] Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 181–196. https://doi.org/10.1145/3385412.3386029

[29] Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.* 2, POPL (2018), 37:1–37:31. https://doi.org/10.1145/3158125

[30] Siddharth Krishna, Alexander J. Summers, and Thomas Wies. 2020. Local Reasoning for Global Graph Properties. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 308–335. https://doi.org/10.1007/978-3-030-44914-8_12

[31] Shuvendu Lahiri and Shaz Qadeer. 2008. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. *SIGPLAN Not.* 43, 1 (jan 2008), 171–182. https://doi.org/10.1145/1328897.1328461

[32] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs Using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (apr 2023), 30 pages. https://doi.org/10.1145/3586037

[33] Oukseh Lee, Hongseok Yang, and Rasmus Petersen. 2011. Program Analysis for Overlaid Data Structures. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 592–608.

[34] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[35] Tal Lev-Ami, Neil Immerman, Thomas Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. 2009. Simulating reachability using first-order logic with applications to verification of linked data structures. *Logical Methods in Computer Science* 5 (04 2009). https://doi.org/10.2168/LMCS-5(2:12)2009

[36] P Lucas. 1968. *Two constructive realizations of the block concept and their equivalence, IBM Lab.* Technical Report. Vienna TR 25.085.

[37] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. 2011. Decidable Logics Combining Heap Structures and Data. *SIGPLAN Not.* 46, 1 (jan 2011), 611–622. https://doi.org/10.1145/1925844.1926455

[38] P. Madhusudan and Xiaokang Qiu. 2011. Efficient Decision Procedures for Heaps Using STRAND. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 43–59.

[39] Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 174 (oct 2022), 30 pages. https://doi.org/10.1145/3563337

[40] Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023. Make Flows Small Again: Revisiting the Flow Framework. In *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings, Part I* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 628–646. https://doi.org/10.1007/978-3-031-30823-9_32

[41] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.).

Springer International Publishing, Cham, 405–425.

[42] Adithya Murali, Lucas Peña, Eion Blanchard, Christof Löding, and P. Madhusudan. 2022. Model-Guided Synthesis of Inductive Lemmas for FOL with Least Fixpoints. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 191 (oct 2022), 30 pages. https://doi.org/10.1145/3563354

[43] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford University, Stanford, CA, USA. AAI8011683.

[44] Greg Nelson and Derek C. Oppen. 1979. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 2 (oct 1979), 245–257. https://doi.org/10.1145/357073.357079

[45] Huu Hai Nguyen and Wei-Ngan Chin. 2008. Enhancing Program Verification with Lemmas. In *Proceedings of the 20th International Conference on Computer Aided Verification* (Princeton, NJ, USA) *(CAV '08)*. Springer-Verlag, Berlin, Heidelberg, 355–369. https://doi.org/10.1007/978-3-540-70545-1_34

[46] Peter W. O'Hearn. 2012. A Primer on Separation Logic (and Automatic Program Verification and Analysis). In *Software Safety and Security - Tools for Analysis and Verification*, Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann (Eds.). NATO Science for Peace and Security Series - D: Information and Communication Security, Vol. 33. IOS Press, 286–318. https://doi.org/10.3233/978-1-61499-028-4-286

[47] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning About Programs That Alter Data Structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*. Springer-Verlag, London, UK, UK, 1–19. http://dl.acm.org/citation.cfm?id=647851.737404

[48] Nisarg Patel, Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2021. Verifying Concurrent Multicopy Search Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 113 (oct 2021), 32 pages. https://doi.org/10.1145/3485490

[49] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. *SIGPLAN Not.* 49, 6 (jun 2014), 440–451. https://doi.org/10.1145/2666356.2594325

[50] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Proceedings of the 25th International Conference on Computer Aided Verification* (Saint Petersburg, Russia) *(CAV'13)*. Springer-Verlag, Berlin, Heidelberg, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

[51] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. Automating Separation Logic with Trees and Data. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV'14)*. Springer-Verlag, Berlin, Heidelberg, 711–728.

[52] Shaz Qadeer. 2023. Boogie Pull Request #669: Monomorphization of polymorphic maps and binders. https://github.com/boogie-org/boogie/pull/669

[53] John C. Reynolds. 1981. *The craft of programming.* Prentice Hall.

[54] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

[55] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

[56] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric Shape Analysis via 3-Valued Logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (may 2002), 217–298. https://doi.org/10.1145/514188.514190

[57] Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2016. Automated Mutual Explicit Induction Proof in Separation Logic. In *FM 2016: Formal Methods*, John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer International Publishing, Cham, 659–676. https://doi.org/10.1007/978-3-319-48989-6_40

[58] Cesare Tinelli and Calogero G. Zarba. 2004. Combining Decision Procedures for Sorted Theories. In *Logics in Artificial Intelligence*, Jóse Júlio Alferes and João Leite (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 641–653.