Predictive Verification using Intrinsic Definitions of Data Structures

ADITHYA MURALI, University of Illinois Urbana-Champaign, Department of Computer Science, USA
CODY RIVERA, University of Illinois Urbana-Champaign, Department of Computer Science, USA
P. MADHUSUDAN, University of Illinois Urbana-Champaign, Department of Computer Science, USA

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 Dafny and prove a suite of data structure manipulating programs correct.

ACM Reference Format:
Adithya Murali, Cody Rivera, and P. Madhusudan. 2023. Predictive Verification using Intrinsic Definitions of Data Structures. 1, 1 (October 2023), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 structure is in such a class or proving that structures in a class have a property is naturally performed using *induction*, typically mirroring the recursive structure in its definition. For example, trees in heaps can be defined using the following recursive definition in first-order logic (FOL) with least fixpoint semantics for definitions:

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

In the above, *htree* maps each location x to the set of all nodes 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 '$\star$' [O'Hearn 2012; O'Hearn et al. 2001; Reynolds 2002a]).

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

Authors' addresses: Adithya Murali, adithya5@illinois.edu, University of Illinois Urbana-Champaign, Department of Computer Science, Urbana, IL, USA; Cody Rivera, codyjr3@illinois.edu, University of Illinois Urbana-Champaign, Department of Computer Science, Urbana, Illinois, USA; P. Madhusudan, madhu@illinois.edu, University of Illinois Urbana-Champaign, Department of Computer Science, Urbana, Illinois, USA.

XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, let alone 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 of first-order logic with least fixpoints and separation logic are both incomplete. Consequently, even though verification frameworks like Dafny [Leino 2010] 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 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 requires engineers to know their internal heuristics and proof mechanisms frustrate verification experience.

Predictable Verification. In this paper, we seek a new kind of *predictable* verification technique, especially for data structure verification. We want a technique where:

(a) the verification engineer declares upfront a promise of what annotations they will provide to prove data structure properties maintained by programs, and

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

The upfront agreement the verification engineer makes on the information they will provide makes their task crystal clear. The fact that the verification engine is decidable 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 in fact wrong. There would be 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.

Intrinsic Definitions of Data Structures. In this paper, we propose an entirely new way of defining structures, called *intrinsic definitions*, that facilitates a predictive verification mechanism for proving maintenance of data structures. Rather than defining data structures using recursion (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 neighbors of each location.

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 $tree : Loc \longrightarrow Bool$, $rank : Loc \longrightarrow \mathbb{Q}^{+}$ (non-negative rationals), and $p : Loc \longrightarrow Loc$ (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 into 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 a 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 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.

Logical underpinnings and Decidable Verification of Annotated Programs. Intrinsic definitions of data structures and the fix-what-you-break program verification methodology are designed carefully in order to ensure decidable verification of annotated programs. When a program starts working with a data structure, the program assumes monadic maps satisfying local conditions exist, which give *local witnesses* for each location that together ensure the global properties of the data structure. When programs destructively update the heap, these witnesses need to be *fixed*, which is what the verification engineer's ghost updates do. However, checking whether the monadic maps on a location have been fixed requires checking only the local property for that location, which is a quantifier-free property.

The verification conditions for Hoare triples involving basic blocks have the following structure. First, the precondition can be captured using *uninterpreted monadic functions* that are assumed to satisfy the local condition on each location that is not in the broken set. 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. Consequently, the entire verification condition is captured in quantifier-free logic involving maps, 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 Dafny. 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.

We consider also *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 combining the instrinsic definitions (maps and local conditions) 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 realize the fix-what-you-break verification methodology in the verification tool Dafny. We introduce the monadic maps as ghost maps, introduce the broken set, and implement updates to the heap using macros that automatically perform updates to the broken set. The user is allowed to assume at any point that any location not in the broken set satisfies the local condition, and be able to remove a location from the broken set after asserting/proving that the local condition holds on it.

We model the above data structures and the annotated methods in Dafny. These annotated programs do not use quantifiers or recursive definitions, and Dafny is able to verify them automatically without further annotations.

Contributions. The paper makes the following contributions:

[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.

- A novel notion of intrinsic definitions of data structures based on ghost monadic maps and local conditions.
- A verification methodology for programs that manipulate data structures with intrinsic definitions following a fix-what-you-break (FWYB) paradigm.
- Intrinsic definitions for several classic data structures, and fix-what-you-break annotations for programs that manipulate such structures.
- A realization of the above intrinsic definitions and program verification in the tool Dafny.

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.

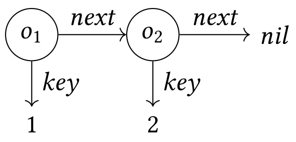
2.1 Data Structures

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 $\mathcal{S} = \{\sigma_0, \sigma_1 \ldots, \sigma_n\}$ and a finite set of fields $\mathcal{F} = \{f_1, f_2 \ldots, f_m\}$. We assume without loss of generality that the sort σ_0 represents 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 $f_i : C \rightarrow \sigma$ 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 $next : C \rightarrow C?$ and a data field $key : C \rightarrow Int$. 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 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 for a fixed arity k. □

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 *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 other interpreted functions over the 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 $\mathit{IDS}(\overline{y})$ 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 $\mathcal{S}$ and 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 [Nelson 1980; Nelson and Oppen 1979; Tinelli and Zarba 2004], containing theories of integers, sets, arrays [de Moura and Bjørner 2009], etc., supported effectively in practice by SMT solvers [Barrett et al. 2011; De Moura and Bjørner 2008].

Definition 2.5 (Data Structures defined by Intrinsic Definitions). Let $C = (\mathcal{S}, \mathcal{F})$ be a class and $\mathit{IDS}(\overline{y}) = (\mathcal{G}, LC, \varphi(\overline{y}))$ 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 $\mathit{SortedLL}(y)$ using monadic maps $\mathit{sortedll} : C \longrightarrow \mathit{Bool}$ and $\mathit{rank} : C \longrightarrow \mathbb{Q}^{+}$ such that:

$$\begin{aligned} LC \equiv\ & \forall x.\ \Big((\mathit{sortedll}(x) \land \mathit{next}(x) \neq \mathit{nil}) \Rightarrow \\ & \qquad (\mathit{sortedll}(\mathit{next}(x)) \land \mathit{rank}(\mathit{next}(x)) < \mathit{rank}(x) \land \mathit{key}(x) \leq \mathit{key}(\mathit{next}(x))) \Big) \\ \varphi(y) \equiv\ & \mathit{sortedll}(y) \end{aligned}$$

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

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).

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 for verifying programs with respect to data structure properties expressed using intrinsic definitions. We begin by describing a while programming language and defining the verification problem we study.

3.1 Programs, Contracts, and Correctness

We fix a class $C = (\mathcal{S}, \mathcal{F})$ throughout this section.

Programs. Figure 1 shows the programming language used in this work. It contains assignments, field lookups, and field mutations, as is usual with programming languages for heaps. We can also use variables and expressions over other sorts. Functions can return multiple outputs. We assume that method signatures contain designated output variables, and therefore the return statement does not mention values. We also note that there is no command for deallocating objects. Instead, we assume that the language has garbage collection. Both these assumptions are true in Dafny where we implement the FWYB methodology (Section 5.1).

Operational Semantics. A program configuration consists of a store (a mapping of variables to values of the appropriate type) and a C-heap. More formally, it is a triple $\theta = (s, O, I)$ where s is a store, O is a finite set of objects of the class C, and I is an interpretation of the function symbols in $\mathcal{F}$. This is similar to the notion of configurations for other programming languages for heaps in prior literature [Kassios 2006; Murali et al. 2023; O'Hearn 2012; Reynolds 2002a]. The operational semantics is the usual one for heap manipulating programs with function calls. In particular, all dereferences must be memory safe and unsafe dereferences lead to the error state

$\perp$. For the remainder of this section, we simply denote the transition between configurations on a program according to the operational semantics by $\theta \xrightarrow{P} \theta'$, and the satisfaction of a pre/post condition on a configuration by $\theta \models \alpha$.

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 functions 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, specifications are of the form "$\overline{w}$ points to a data structure *IDS* such that the (quantifier-free) property $\psi(\overline{z})$ holds". For example, given inputs x of type C and k of type *Int* to a program we can specify a precondition that x points to a sorted linked list such that the key stored at x is smaller than k. The corresponding formula is $\exists\, sortedll, rank.\ (LC \wedge sortedll(x) \wedge key(x) < k)$, where LC is the local condition for a sorted linked list defined in Example 2.6.

We study the validity of the following Hoare Triples:

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

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

We use a simple contract for insertion into a sorted linked list as a running example:

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

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

where x, r are variables of type C, k is of type *Int* and *sorted–insert* is the usual recursive method that inserts a key into a sorted linked list by recursively traversing the list starting from x until it finds the appropriate place to insert k. We provide the relevant snippets to explain our methodology in later sections.

Note that the input x is of type C (rather than $C?$) and therefore it cannot be *nil*. We do this for simplicity of exposition; one can write a more complex contract allowing for x possibly being *nil*.

Validity of Intrinsic Hoare Triples. We define the validity of Hoare Triples using the notation developed above:

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 $\theta \xrightarrow{P} \theta'$ for some θ', 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 from the monadic maps to obtain specifications of the form $LC \wedge \varphi \wedge \psi$. We do this by re-framing the problem such that the g_i maps are explicitly available in the pre state as "ghost" fields of objects (such that they satisfy $LC \wedge \varphi \wedge \alpha$), and requiring the verification engineer to update the fields using *ghost code* and construct the g_i maps for the post state such that they satisfy $LC \wedge \varphi \wedge \beta$. Intuitively, instead of reasoning with specifications that say there exist maps (obtained magically!) in the pre and post state that satisfy certain conditions, the verification engineer argues that for *any* maps given in the pre state satisfying the precondition, we can *compute* a set of corresponding maps for the post state satisfying the postcondition. This transforms the problem of study into 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 values of the fields in $\mathcal{G}$ wherever the execution of P breaks the local conditions LC.

In Stage 2 (Section 3.3) we partially remove the first-order quantification in specifications by restricting the set of objects where local conditions are required to hold. Specifically, we introduce a special variable called the *broken set Br* and use it to track objects where the program's changes to the heap destroy the local conditions. Then we only require that at any point in the program, local conditions hold for objects not in the broken set. Formally, we transform specifications of the form $(\forall x.\, \rho) \wedge \varphi \wedge \psi$ where $LC \equiv \forall x.\, \rho$ into $(\forall x \notin Br. \rho) \wedge \varphi \wedge \psi$[2]. We require the verification engineer to maintain the Br set accurately by adding objects to it when the program breaks their local conditions and removing them when the engineer's updates to their ghost fields fixes the local conditions on them. This leaves us with triples of the form $\langle\, (\forall x \notin Br) \wedge \varphi \wedge \alpha \,\rangle\; P_{\mathcal{G},Br} \;\langle\, (\forall x \notin Br) \wedge \varphi \wedge \beta \,\rangle$ where $P_{\mathcal{G},Br}$ is an augmentation of P with ghost code that updates both $\mathcal{G}$ and Br correctly.

In Stage 3 (Section 3.4) we show how to remove the quantification entirely for programs with *well-behaved* manipulations of the broken set, reducing the problem to triples of the form $\langle\, \varphi \wedge \alpha \,\rangle\; P_{\mathcal{G},Br} \;\langle\, \varphi \wedge \beta \,\rangle$. Intuitively, the well-behaved programming paradigm ensures careful handling of the broken set by forcing the verification engineer to add objects to the broken set if their local conditions are potentially broken by statements that change the heap (mutations, allocation, etc.) and only allowing them to remove the objects from the broken set if the engineer can show that local conditions hold on them. Well-behavedness is syntactically checkable and forms a class capable of expressing many data structure manipulation methods, as shown by our case studies and evaluation (see Section 4 and Section 5.3).

One of the salient features of monadic maps and local conditions is that they offer simple *frame reasoning*. When the fields of a location x are mutated, the local condition may cease to hold for x as well as neighbors of x (locations accessible from x using pointer-sequences or locations that can access x using pointer-sequences, where these pointer-sequences are mentioned in the local condition). However, we know that local conditions are not affected on other locations further away, and we exploit this implicit frame reasoning to maintain broken sets.

In Section 3.5 we prove the soundness of the entire FWYB methodology, namely that if $\langle\, \varphi \wedge \alpha \,\rangle\; P_{\mathcal{G},Br} \;\langle\, \varphi \wedge \beta \,\rangle$ is valid and $P_{\mathcal{G},Br}$ is well-behaved, then the triple $\langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \alpha) \,\rangle\; P \;\langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \wedge \varphi \wedge \beta) \,\rangle$ intended by the user is valid. The validity of the former triple can be discharged effectively by decision procedures for combinations of quantifier-free theories, which makes the FWYB methodology entirely automatic given the ghost annotations.

[2]This is of course more general, as setting $Br = \emptyset$ degenerates to the original specifications. We need the more general form to state contracts of called methods and loop invariants at a general point in the program where the set of broken objects may not be empty.

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

Consider an intrinsic Hoare Triple $\langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \land \varphi \land \alpha)\, \rangle\; P\; \langle\, \exists\, g_1, g_2 \ldots, g_k.\, (LC \land \varphi \land \beta)\, \rangle$. 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 $LC \land \varphi \land \alpha$, and we require the verification engineer to *compute* the $\{g_i\}$ maps in the post state satisfying $LC \land \varphi \land \beta$. 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 [Filliâtre et al. 2016; Jones 2010; Lucas 1968; Reynolds 1981]. Let us consider an example:

Example 3.3 (Proof using Ghost Code). We use the running example (Example 3.1) of insertion into a sorted list. Figure 2 shows on the left a snippet where the key k that is 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.

```
pre:  ∃ sortedll, rank. LC ∧ sortedll(x)
post: ∃ sortedll, rank. LC ∧ sortedll(x)
 y := x.next;
 z := new C();
 z.key := k;
 z.next := y;
 x.next := z;
```

```
pre:  LC ∧ sortedll(x)
post: LC ∧ sortedll(x)
 y := x.next;
 z := new C();
 z.key := k;
 z.next := y;
 z.sortedll := True;
 x.next := z;
 z.rank := (x.rank + y.rank)/2;
 // LC holds for x,y, and z
```

Fig. 2. Left: Code and specifications written by the user; Right: Augmented program with ghost updates. Specifications do not quantify over monadic maps.

Let us assume that objects now have fields *sortedll* and *rank* satisfying the local conditions in the pre state. The local conditions (see Example 2.6) say that if *sortedll* holds on x (which is true in our case), then *sortedll* must also hold on $next(x)$, the *rank* of $next(x)$ must be smaller than the *rank* of x, and its key must be larger than the key of x. The program violates some of these conditions.

Observe that the required order on keys is already satisfied since the insertion is correct. The conditions that are violated are: (a) $sortedll(z)$ must hold since $sortedll(x)$ holds, (b) the rank of z must larger than that of y, and (c) the rank of x must be larger than that of z. We fix these violations using ghost updates, shown in Figure 2 on the right (marked in blue).

Here the specifications do not contain the second-order existential quantification over the monadic maps. Instead, they are treated as (ghost) fields and manipulated using the usual field update statements.

A formal proof of the sufficiency of the repairs shown above is complex. In general, the mutation `x.next := z` may break both `x` and `z`, similarly each field update has its own set of *impacted objects*, and one must reason about a set of such updates and fixes. Our contributions in Stage 2 and Stage 3 presented in later sections enable the engineer to systematically track broken objects and reason about the sufficiency of repairs effectively. □

We note a point of subtlety here: the second triple in the above example eliminates existential quantification over $\mathcal{G}$ by claiming something stronger than the original specification, namely that for *any* maps *sortedll* and *rank* such that $LC \wedge sortedll(x)$ is satisfied in the pre state, there is a *computation* that yields corresponding maps in the post state such that the property is preserved. The onus of coming up with such a computation is placed on the verification engineer.

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. The key difference between ghost fields and other fields is that syntactically valid programs cannot compute values of non-ghost variables/fields using ghost fields, and branching conditions in conditional expressions or loops must be non-ghost expressions.

Performing the above transformation 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 3.4. *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*[3]*, where P is the projection of $P_{\mathcal{G}}$ obtained by eliminating ghost code.*

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

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

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

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

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.

Note the conjunct $Br = \emptyset$ in the pre and post conditions for P, which make them equivalent to the original specifications. Indeed, for the main method that preserves the data structure property,

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

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 of this transformation follows from an argument similar to the correctness of Proposition 3.4.

The above transformation turns the problem of verifying a complex program with multiple auxiliary methods into a modular verification problem where each method has a contract of the same form. We describe how to exploit this uniformity in specifications to achieve decidable reasoning in the next section.

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

We consider triples of the form

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

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

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

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 given by Stage 2 to be dropped since it always holds for a well-behaved program. Let us look at such a program in the context of our running example:

Example 3.5 (Well-Behaved Sorted List Insertion). We first relax the universal quantification as described in Stage 2 (Section 3.3) and rewrite the pre and post conditions to $\forall z \notin Br.\, LC(z)) \wedge$ $sortedll(x) \wedge Br = \emptyset$.

We then write the following well-behaved augmentation of the original program. 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 that are enforced by the well-behavedness paradigm in pink and the ghost updates written by the verification engineer in blue. As we can see, the paradigm adds the set of 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. We also see that if the engineer wants to remove x from the broken set then they are required to show that $LC(x)$ holds (assert followed by removal from Br). Finally, as seen at the beginning of the program, if we show x does not belong to Br then well-behavedness allows us to infer that $LC(x)$ holds. This is possible because well-behaved programs always maintain the broken set correctly. We leave the formal correctness of the above program with the quantifier-free specifications to the reader.

Putting it All Together. Let us call the above program $P_{\mathcal{G},Br}$. 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 $\langle\, (\forall z \notin Br.\, \rho) \wedge sortedll(x) \wedge Br = \emptyset \,\rangle\ P_{\mathcal{G},Br}\ \langle\, (\forall z \notin Br.\, \rho) \wedge sortedll(x) \wedge Br = \emptyset \,\rangle$. By Proposition 3.4, this in turn means that the triple with the user's original program given in Example 3.3 with existential quantification over monadic maps and universal quantification over objects in the specifications is valid! Therefore, via FWYB we can reason about the correctness of programs with respect to intrinsic specifications by checking the correctness of (augmented) programs with respect to the quantifier-free specifications, which can be discharged efficiently in practice using SMT solvers which support decision procedures for combinations of theories [Barrett et al. 2011; De Moura and Bjørner 2008]. □

We now develop 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 consisting of augmented programs and quantifier-free annotations by $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, with $\{\}$ brackets rather than $\langle\,\rangle$. We denote that such a triple is provable by $\vdash \{\psi_{pre}\}\ P\ \{\psi_{post}\}$. Our theory is agnostic to the mechanism for proving such Hoare triples correct (and in evaluations, we use the off-the-shelf verification tool Dafny). However, we assume that the underlying proof mechanism is sound with respect to the operational semantics, checking in particular that dereferences are memory safe. We denote that a code snippet P is well-behaved by $\vdash_{\mathrm{WB}} P$. We also denote that local conditions hold on an object x by $LC(x)$ for clarity (rather than the technically correct $\rho(x)$ where ρ is the matrix of LC).

The rules for constructing well-behaved programs are given in Figure 3.

Skip, assignment, lookup, and return statements are vacuously well-behaved. They do not change the heap and therefore preserve the local condition on any object where they already held.

Mutation. Mutation statements potentially break local conditions and must therefore grow the broken set. Let A be a finite set of location 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 the broken set after a mutation with its impact set (see mutation statements in Example 3.3 for instances of this rule). Of course, the impact set may not always be expressible as a finite set of terms, but this is indeed the case for all intrinsic definitions of data structures that we know define in this paper. We show how to construct provably correct impact sets for field mutations in practice in Section 4.1.

Allocation. An allocation statement 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 3.5).

Function Call. We state axiomatically that function call statements are well-behaved, leaving it to the verification engineer to capture the contract of the called function accurately, including its effect on Br.

Assert LC and Remove. The rules described above grow the broken set. We also want to shrink the broken set when the verification engineer fixes the local conditions on some broken locations. The Assert LC and Remove rule enables this. This rule is used in Example 3.5 as the sequence of statements `assert LC(x); Br := Br\{x}`. Informally, the verification engineer is required to show that $LC(x)$ holds before removing x from Br.

Infer LC Outside Br. We also add a rule for instantiating LC on objects outside the broken set. Since our methodology makes the conjunct $\forall x \notin Br.\, \rho(x)$ implicit, we lose some proving power. For example, if a program modifies a location x in a red black tree, we may want to update the values of the ghost monadic maps on x depending on whether its children are red or black, and using the information given by the local constraints on its children (since the children are not broken and therefore satisfy the local constraints). The Infer LC Outside Br rule enables this reasoning mechanism. We employ this rule in Example 3.5 using the sequence of statements `assert x∉Br; assume LC(x)`.

Finally, compositions, conditionals, and loops involving well-formed subprograms are well-formed. Note that Br is a ghost variable and therefore cannot appear in the conditions for conditional statements or loops.

In the above presentation we use only one broken set for simplicity of exposition. However, our general framework allows for finer-grained broken sets that can track breaks over a partition on the local conditions. For example, Section 4.6 illustrates verifying deletion in an overlaid data structure (consisting of a linked list and a binary search tree) using two broken sets: one each corresponding to the local conditions of the two component data structures.

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 M_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 3.6 (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:

$$\textsc{Skip/Assignment/Lookup/Return}\quad \frac{}{\vdash_{\mathrm{WB}} s \text{ where } s \text{ is of the form} \atop \texttt{skip, x:=y, x:=y.f, or return}}$$

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

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

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

$$\textsc{Infer LC Outside Br}\quad \frac{}{\vdash_{\mathrm{WB}}\ \mathsf{if}\ x \notin Br\ \mathsf{then\ assume}\ LC(x)\ \mathsf{else\ skip}}$$

$$\textsc{Assert LC and Remove}\quad \frac{}{\vdash_{\mathrm{WB}}\ \mathsf{if}\ LC(x)\ \mathsf{then}\ Br := Br \setminus \{x\}\ \mathsf{else\ skip}}$$

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

$$\textsc{If-Then-Else}\quad \frac{\vdash_{\mathrm{WB}}\ P \qquad \vdash_{\mathrm{WB}}\ Q}{\vdash_{\mathrm{WB}}\ \mathsf{if}\ cond\ P\ \mathsf{else}\ Q \atop \text{where } cond \text{ does not mention } Br}$$

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

Fig. 3. Rules for constructing well-behaved programs. Local condition formula instantiated at x is denoted by $LC(x)$.

(1) If M has signature $(\overline{x}, Br,\ ret : \overline{y}, Br)$, then M' has signature $(\overline{x},\ ret : \overline{y})$. 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 3.7 (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 3). 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 3.6. 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 A.

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:

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

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 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 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 only 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 4 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 affect objects that are far away on the heap.*

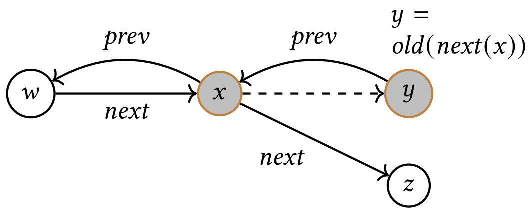


Fig. 4. Reasoning about the set of objects broken by `x.next := z`. The dashed arrow represents the old *next* pointer before the mutation. The grey nodes denote objects where local conditions can be broken by the mutation. We see that only x and y may violate *next* and *prev* being inverses.

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*.

In our case, we claim that impact set contains at most x and $old(next(x))$. Here's a proof (see Fig 4): 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 3 characterizes the impact set A_f for mutation on a field f thus:

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

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 DAFNY (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 3). We can realize these restrictions using macros:

(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 A_f(x)`. Here `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.

```
pre: Br = ∅
post: LC(r) ∧ prev(r) = nil
      ∧ Br = ite(old(prev(x)) = nil, ∅, {old(prev(x))})
      ∧ length(r) = old(length(x)) + 1
      ∧ keys(r) = old(keys(x)) ∪ {k}
      ∧ old(hslist(x)) ⊂ hslist(r)
modifies: hslist(x)
sorted_list_insert(x: C, k: Int, Br: Set(C))
returns r: C, Br: Set(C)
{
  InferLCOutsideBr(x, Br);
  if (x.key ≥ k) then { // k inserted before x
    NewObj(z, Br);          // {z}
    Mut(z, key, k, Br);    // {z} since z.prev = nil
    Mut(z, next, x, Br);   // {z} since z.next = nil
    Mut(z, hslist, {z} ∪ x.hslist, Br); // {z}
    Mut(z, length, 1 + x.length, Br);    // {z}
    Mut(z, keys,  {k} ∪ x.keys, Br);    // {z}
    Mut(x, prev, z, Br);     // {z, x, old(prev(x))}
    AssertLCAndRemove(z, Br);  // {x, old(prev(x))}
    AssertLCAndRemove(x, Br); // {old(prev(x))}
    r := z;
  }
  else {
  if (x.next = nil) then { // one-element list
    NewObj(z, Br);
    Mut(z, key, k, Br);
    Mut(z, next, nil, Br);
    Mut(z, hslist, {z}, Br);
    Mut(z, length, 1, Br);
    Mut(z, keys, {k}, Br);
    Mut(x, next, z, Br);
    Mut(z, prev, x, Br);
    AssertLCAndRemove(z, Br);
    Mut(x, prev, nil, Br);
    Mut(x, hslist, {x} ∪ {z}, Br);
    Mut(x, length, 2, Br);
    Mut(x, keys, {x.key} ∪ {k}, Br);
    AssertLCAndRemove(x, Br);
    r := x;
  }
  else { // recursive case
    y := x.next;
    InferLCOutsideBr(y, Br);
    tmp, Br := sorted_list_insert(y, k, Br);   // {x}
    InferLCOutsideBr(y, Br);
    if (y.prev = x) then {
      Mut(y, prev, nil, Br);     // {y, x}
    }
    Mut(x, next, tmp, Br);       // {y, x}
    AssertLCAndRemove(y, Br);    // {x}
    Mut(tmp, prev, x, Br);       // {tmp, x}
    AssertLCAndRemove(tmp, Br); // {x}
    Mut(x, hslist, {x} ∪ tmp.hslist, Br);  // {x, prev(x)}
    Mut(x, length, 1 + tmp.length, Br);    // {x, prev(x)}
    Mut(x, keys, {x.key} ∪ tmp.keys, Br);  // {x, prev(x)}
    Mut(x, prev, nil, Br);        // {x, old(prev(x))}
    AssertLCAndRemove(x, Br);    // {old(prev(x))}
    r := x;
  }}
}
```

Fig. 5. Code for insertion into a sorted list written in the syntactic fragment for well-behaved programs(Section 4.1). Black lines denote code written by the user, and blue lines denote lines written by the verification engineer. The comments on the right show the state of the broken set *Br* after the statement on the corresponding line.

(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 ∉ 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.

Verifying Sorted List Insertion. We provide the specifications and the code augmented with ghost annotations in Figure 5.

Specifications. The precondition states that the broken set is empty at the beginning of the program. The postcondition states that the returned object r satisfies the local conditions and satisfies the correlation formula for a sorted list (i.e., $prev(r) = nil$). However, the broken set is only empty if the input object x was the head of a sorted list, and it is $\{prev(x)\}$ otherwise. The other conjuncts express functional specifications for insertion in terms of the length, heaplet, and set

[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).

of keys. We also add a 'modifies' clause which enables program verifiers for heap manipulating programs (including Dafny) to utilize frame reasoning across function calls.

Summary. The proof works at a high-level as follows: we recurse down the list, reaching the appropriate object x before which the new key must be inserted. This is the first branch in Figure 5, and we show the broken set at each point in the comments to the right. We create the new object z with the appropriate key and point $z.next$ to x. We then fix the local conditions on x and z. However, these fixes break the LC on $old(prev(x))$. We maintain this property up the recursion, at each point fixing LC on x and breaking it on $old(prev(x))$ in the process. This is shown in the last branch in the code. We eventually reach the head of the sorted list, whose $prev$ in the pre state is nil, and at that point the fixes do not break anything else, i.e., the broken set is empty (as desired).

The verification engineer adds ghost code to perform these fixes as shown in blue in Figure 5. We can also see that there are essentially as many lines of ghost code as there are lines of user code; we compare these values across our benchmark suite (see Table 2) and find that this is typical for many methods. However, the verification conditions for the (augmented) program are *decidable* because they can be stated using quantifier-free formulas over decidable combinations of theories including maps, map updates, and sets.

4.2 BST Right-Rotation

We now turn to another data structure and method that illustrates intrinsic definitions for trees, namely verifying a right rotate on a binary search tree. Such an operation is a common tree operation, and rotations are used widely in maintaining balanced search trees, such as AVL and Red-Black Trees, on which several of our benchmarks operate.

We augment the definition of binary trees discussed in Section 1 to include the $min : BST \longrightarrow Real$ and $max : BST \longrightarrow Real$ maps, which capture the minimum and maximum keys stored in the tree rooted at a node, to help enforce binary search tree properties locally. The local condition and the impact sets are as below:

$$\begin{aligned}
LC \equiv\ & \forall x.\, min(x) \le key(x) \le max(x)\\
& \wedge (p(x) \neq nil \Rightarrow l(p(x)) = x \vee r(p(x)) = x)\\
& \wedge (l(x) = nil \Rightarrow min(x) = key(x))\\
& \wedge (l(x) \neq nil \Rightarrow p(l(x)) = x \wedge rank(l(x)) < rank(x)\\
& \qquad\qquad \wedge\ max(l(x)) < key(x)\ \wedge\ min(x) = min(l(x)))\\
& \wedge (r(x) = nil \Rightarrow max(x) = key(x))\\
& \wedge (r(x) \neq nil \Rightarrow p(r(x)) = x \wedge rank(r(x)) < rank(x)\\
& \qquad\qquad \wedge\ min(r(x)) > key(x)\ \wedge\ max(x) = max(r(x)))
\end{aligned}$$

Mutated Field f	Impacted Objects A_f
l	$\{x, old(l(x))\}$
r	$\{x, old(r(x))\}$
p	$\{x, old(p(x))\}$
key	$\{x\}$
min	$\{x, p(x)\}$
max	$\{x, p(x)\}$
$rank$	$\{x, p(x)\}$

An annotated text of the procedure is presented in Appendix B, but we present the gist of how the data structure is repaired here. Recall that in a BST right rotation, that there are two nodes x and y such that y is x's left child. After the rotation is performed, y becomes the new root of the subtree, while x becomes y's right child. Several routine updates of the monadic map p (parent) will have to be made. The most interesting update is that of the $rank : BST \longrightarrow Real$ map. Since y is now the root of the affected subtree, its rank must be greater than all its children. One way of doing this is to increase y's rank to something greater than x's rank. This works if y has no parent, but not in general. To solve this issue, we use the density of the Reals to set the rank of y to $(rank(x) + rank(p(y)))/2$. Note that there are a fixed number of ghost map updates, as the various monadic maps for distant ancestors and descendents of x, y do not change (the min/max of subtrees of such nodes do not change).

4.3 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:

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

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. 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.

4.4 Merging Sorted Lists

We demonstrate the ability of intrinsic definitions to handle multiple data structures at once, using the example of in-place merging of two sorted lists. The method merges the two lists by reusing the two lists' elements, which is a natural pattern for imperative code. Once again, we extend the definition of sorted lists from Case Study 4.1. We add the predicates $list1 : C \longrightarrow Bool$, $list2 : C \longrightarrow Bool$, and $list3 : C \longrightarrow Bool$, to indicate disjoint classes of lists. The local condition and impact sets are:

$$\begin{aligned}
&\ (list1(x) \vee list2(x) \vee list3(x)) \\
&\wedge \neg(list1(x) \wedge list2(x)) \wedge \neg(list2(x) \wedge list3(x)) \\
&\wedge \neg(list1(x) \wedge list3(x)) \\
&\wedge (list1(x) \Rightarrow (next(x) \neq nil \Rightarrow list1(next(x)))) \\
&\wedge (list2(x) \Rightarrow (next(x) \neq nil \Rightarrow list2(next(x)))) \\
&\wedge (list3(x) \Rightarrow (next(x) \neq nil \Rightarrow list3(next(x))))
\end{aligned}$$

Mutated Field f	Impacted Objects A_f
$list1$	$\{x, prev(x)\}$
$list2$	$\{x, prev(x)\}$
$list3$	$\{x, prev(x)\}$

Disjointness of the three lists is ensured by insisting that every object has at most one of the three list predicates hold.

We give a gist of the proof of the merge method. The recursive program compares the keys at the heads of the first and second sorted lists, and adds the appropriate node to the front of the third list. It turns out that we can easily update the ghost maps for this node (making it belong to the third list, and updating its parent pointer and key set) as well as updating the parent pointer of the head of the list where the node is removed from. When one of the lists is empty, we append the third list to the non-empty list using a single pointer mutation and then, using a ghost loop, we update the nodes in the appended list to make $list3$ true (this needs a loop invariant involving the broken set).

4.5 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 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:

$$\begin{aligned}&(x = last(x) \Rightarrow last(next(x)) = x \land length(x) = 0 \land rev\_length(x) = 0)\\ \land\ &(x \neq last(x) \Rightarrow last(next(x)) = last(x) \land length(x) = length(next(x)) + 1\\ &\qquad \land rev\_length(x) = rev\_length(prev(x)) + 1)\end{aligned}$$

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 4.3, we make appropriate updates to the $length$ and $keys$ maps, which are not fully described here, following the $prev$ map until we reach $last(x)$.

4.6 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 [Lee et al. 2011]. 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 [Lee et al. 2011], 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 $list_{\mathrm{head}}$ 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 $list_{\mathrm{head}}$, 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:

$$\begin{aligned} Valid \equiv\ & bst\_root(h) = r \land list\_root(r) = h \\ & \land\ list\_heaplet(h) = bst\_heaplet(r) \end{aligned}$$

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.

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We demonstrate the technique of intrinsic definitions of data structures and FWYB verification by implementing them in the programming language/program verifier Dafny [Leino 2010].

We had two choices— one was to implement a custom verifier that generated verification conditions directly to an SMT solver or to work with the mature Dafny-Boogie toolchain. One disadvantage of working with Dafny is that it has its own memory model and verification condition generation for it (using Boogie) that introduces quantified formulas even when specifications are quantifier-free. However, we chose the Dafny framework for the advantages it gave— a fairly realistic object-oriented language that can be compiled and executed and an advanced verification framework with in-built frame reasoning (with respect to modifies clauses). Implementing in Dafny also immediately makes our technique available to the growing set of Dafny users to prove their programs correct, where they can combine traditional inductive verification as well as verification using intrinsic definitions. We did find that Dafny's performance was less than what we expected in some of the benchmarks, given that the verification conditions could fall in decidable SMT theories. We leave optimizing Dafny for handling intrinsic definitions more efficiently to future work (see Section 7).

5.1.1 Dafny support for Fix-What-You-Break Verification. The main support to verify programs is to implement the *macros* defined in Section 4.1: `Mut`, `NewObj`, `AssertLCAndRemove` and `LCOutsideBr`. This allows programming in the discipline that ensures broken sets are maintained correctly.

We propose methods for mutation `Mut`, where mutation of each field is parameterized by an impact set (proving the correctness of the impact set is done independently using Dafny). We also implement the method for `NewObj` to allocate a new object and add it to the broken set. To implement `AssertLCAndRemove`, we create a method that requires that the local condition holds on what you pass to it, and then ensures that the node is removed from the returned broken set. Finally, to implement `LCOutsideBr`, we define an axiom such that if something is not in the broken set, one may assert that its local condition holds.

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. Additionally, we include 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 (from [Lee et al. 2011]). The contracts for these functions are fairly 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 resulting data structure.

5.3 Evaluation

We evaluate our technique by implementing the benchmarks in Dafny. We implement the code using the macros and the specification using monadic maps and local conditions, strengthening contracts for functions using assertions on broken sets. We then insert ghost code that repairs the monadic maps on locations to remove them from the broken set, and meet the contract. As we have articulated in the previous, the intrinsic definitions and the updates to monadic maps require a new way of thinking about programs and repairs.

Our verification efforts are detailed in the table in Table 2, for 44 methods over 10 data structure definitions.

Dafny can be used in two configurations to handle sets/arrays that we extensively use (broken sets, sets of keys, heaps, etc.). The first is where it uses its default mode handling arrays/sets using a set of axioms that is often enough, and the second, where one can ask it (and Boogie) to use the underlying Z3 theory of arrays. However, for the latter, the Boogie programs need to *monomorphize* several definitions that are modeled polymorphically, and this translation does not succeed always, though there have been new versions of Boogie that monomorphize and use Z3's array theory more reliably [Qadeer 2023]. Verification times vary between the modes depending on the program. We hence execute both modes, and report in the table the *lower* of the times (the idea being that we run both modes in parallel, and exit when either of them succeeds). As mentioned above, optimizing Dafny for intrinsic definitions and exploiting the decidable VCs generated is left to future work.

All benchmarks were verified by Dafny in reasonable time. Notice that the lines of ghost code written is nontrivial, but these are typically simple involving computationally repairing monadic maps and manipulating broken sets. We believe that many of these annotations can be automated in future work using program synthesis techniques (see Section 7). Note, however, that none of these programs required further annotations in terms of instantiations, triggers, inductive lemmas, etc. in order to prove them correct.

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 incomplete logic (where validity is not even recursively enumerable), but support effective strategies nevertheless, using heuristics, lemma synthesis, and further annotations from the programmer [Banerjee et al. 2008a; Banerjee and Naumann 2013; Banerjee et al. 2008b, 2013; Berdine et al. 2005b, 2006; Calcagno et al. 2011; Chin et al. 2007; Chu et al. 2015; Distefano et al. 2006; Jacobs et al. 2011; Murali et al. 2022; Nguyen and Chin 2008; O'Hearn 2012; Pek et al. 2014; Piskac et al. 2014; Reynolds 2002a; Ta et al. 2016]. In this paper, we have proposed a new paradigm of predictive verification that calls for the programmer to write a reasonable amount of extra annotations (which in our case is ghost code that updates monadic maps) 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 [Lev-Ami et al. 2009], and the logic Lisbq in the Havoc

Data Structure	Lines of *LC*	Impact Set Verif. Time	Method	Lines of Spec.	Lines of Annotation	Lines of Code	Verif. Time
Singly-Linked List	13	3s	Append	22	10	8	5s
			Copy-All	11	10	9	4s
			Delete-All	12	20	16	4s
			Find	6	3	7	3s
			Insert-Back	11	12	10	5s
			Insert-Front	9	7	5	3s
			Insert	11	20	16	5s
			Reverse	10	23	8	6s
Sorted List	20	4s	Delete-All	14	20	15	5s
			Find	7	4	7	3s
			Insert	13	29	17	6s
			Merge	21	24	13	6s
			Reverse	11	32	8	5s
Sorted List (w. *min*, *max* maps)	25	5s	Concatenate	24	15	10	8s
			Find-Last	8	8	5	3s
Circular List	43	4s	Insert-Front	6	31	7	4s
			Insert-Back	7	32	11	4s
			Delete-Front	7	31	6	3s
			Delete-Back	8	32	9	3s
Binary Search Tree	36	5s	Find	8	10	7	4s
			Insert	20	38	19	16s
			Delete	24	40	22	27s
			Remove-Root	18	39	32	1m53s
Treap	38	5s	Find	8	10	7	3s
			Insert	24	74	37	9m20s
			Delete	26	40	22	32s
			Remove-Root	24	66	47	3m42s
AVL Tree	102	6s	Insert	22	36	22	58s
			Delete	32	60	74	4m30s
			Find-Min	5	6	5	3s
			Balance	20	98	76	9m21s
Red-Black Tree	157	7s	Insert	25	191	112	2h17m
			Delete	36	65	91	5m5s
			Find-Min	5	6	5	3s
			Del-L-Fixup	24	99	59	15m16s
			Del-R-Fixup	24	99	59	19m
BST+Scaffolding	167	6s	Delete-Inside	26	52	3	40s
			Remove-Root	33	57	52	15m48s
			Fix-Depth	15	12	0	4s
Scheduler Queue (overlaid SLL+BST)	200	8s	Move-Request	15	10	7	5s
			List-Remove-First	20	10	6	8s
			BST-Delete-Inside	32	55	3	1m17s
			BST-Remove-Root	40	61	52	2h15m
			BST-Fix-Depth	18	14	0	5s

Table 2. Implementation and Verification of Dafny programs on the benchmarks. The columns give data structure, lines of code in defining local conditions, time to verify impact set for all pointer mutations in the data structure, the method, number of lines of specification (pre/post) and ghost code annotations (invariants/monadic map updates), lines of code of the original program, and the verification time.

tool [Lahiri and Qadeer 2008] offer decidability. There are also several decidable fragments of separation logic known [Berdine et al. 2005a; Piskac et al. 2013] as well as fragments that admit a decidable entailment problem for them [Echenim et al. 2021]. Decidable logics based on interpreting bounded treewidth data structures on trees have also been studied, for separation logics as well as other logics [Iosif et al. 2013; Madhusudan et al. 2011; Madhusudan and Qiu 2011]. However, in general, these logics are heavily restricted— the magic wand in separation logic quickly leads to undecidability [Brochenin et al. 2008], the general entailment problem for separation logic with inductive predicates is undecidable [Antonopoulos et al. 2014], 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. A crucial aspect of intrinsic definitions is the use of monadic maps to capture data structures and local properties to constrain them. While we do not know of any work that uses repairs of such maps as we do to facilitate verification, monadic maps have been explored in some earlier works. First, in shape analysis [Sagiv et al. 2002], an abstract interpretation framework for heaps, monadic predicates are often used to express inductively defined properties of single locations on the heap (for example, for a fixed node h, a predicate $sorted\_lseg\_h(x)$ could capture whether h to x forms a sorted list segment). In shape analysis, these predicates are useful to differentiate between locations as all locations that satisfy the same monadic properties are merged into a single abstract location in order to define finite-space abstractions.

In separation logic, the *iterated separating conjunction operator*, introduced very early already by Reynolds in 2002 [Reynolds 2002b], 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 [Distefano and Parkinson 2008; Müller et al. 2016]. The work on verification using flows [Krishna et al. 2018, 2020] 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 on updating these maps or showing subsequent decidable reasoning.

Ghost code. The methodology of writing ghost code is a common paradigm in deductive program verification [Filliâtre et al. 2016; Jones 2010; Lucas 1968; Reynolds 1981] and supported by verification tools such as Dafny [Leino 2010, 2023]. Ghost code involves code that manipulate auxiliary variables to perform a parallel computation with the original code without affecting it (conditionals on ghost code that update original code variables are prohibited, etc.). Intuitively, ghost code computes a state that is useful for verification of assertions/post-conditions. One typical use case is when verification demands a certain map to exist, and where the ghost code *computes* this map as the original code evolves. For example, a method may start with a concrete data structure that refines an abstract datatype witnessed by a refinement map, and the programmer can establish a new refinement map at the exit of the method by constructing the refinement map in tandem with the original code. 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

Intrinsic definitions eschew recursion/induction and instead define data structures using monadic maps and local conditions. To prove that a program maintains a valid data structure hence requires only maintaining monadic maps and verifying the local conditions involving them on locations that get broken. This facilitates verification using ghost-code annotations that allow the verification engineer to constructively maintain monadic maps to prove programs correct. Furthermore, verifying that engineer-provided ghost code annotations are indeed correct falls in decidable theories, leading to a predictable verification framework. Verification engineers hence are asked to provide annotations that establish monadic maps, and in turn are assured decidable verification that saves them from having to give further tactics, heuristics, inductive lemmas/strengthenings, etc. that are typically needed in current verification paradigms.

Future Work. First, it would be useful to develop versions of tools (such as Dafny) that have native support for intrinsic definitions and verify programs using SMT solvers in more efficient ways, exploiting the fact that the verification conditions fall in decidable fragments. 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, we believe many of the ghost code annotations we write are fairly simple and can be *automated using program synthesis techniques* (especially updating ghost field pointers on a location in order to fix it to meet the local conditions). An interactive system that suggests fixes to programmers as they annotate code would reduce annotation burden. 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 in the context of concurrent and distributed programs that maintain data structures is particularly interesting. Finally, we believe that intrinsic definitions opens up an entirely new approach to defining properties of structures. A theoretical study of the expressive power of intrinsic definitions and connections to logics studied in finite model theory would be interesting.

REFERENCES

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.

Anindya Banerjee, Mike Barnett, and David A. Naumann. 2008a. 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.

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

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

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

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.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005a. 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.

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

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

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.

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

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

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

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.

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

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.

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

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.

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

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.

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.

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

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer-Verlag, Berlin, Heidelberg, 268–283.

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

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

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

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.

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.

Rustan Leino. 2023. *Programming Proofs*. The MIT Press.

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

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

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

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.

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.

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

Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. 2023. A First-Order Logic with Frames. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 7 (may 2023), 44 pages. https://doi.org/10.1145/3583057

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

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

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

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

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

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

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

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.

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

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

John C. Reynolds. 2002a. 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.

John C. Reynolds. 2002b. 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.

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

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

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.

A CONTINUED FROM SECTION 3.5

We provide here the gist of a proof of Theorem 3.7, which states that the FWYB methodology is sound. We assume that there is only one method for simplicity.

Fix an intrinsic definition $(\mathcal{G}, LC, \varphi)$ where $\mathcal{G} = \{g_1, g_2 \ldots, g_l\}$ and $LC \equiv \forall z.\, \rho(z)$. Let M be a method and $P_{\mathcal{G},Br}$ be its body such that $\vdash_{\text{WB}} P_{\mathcal{G},Br}$. Let ψ_{pre} and ψ_{post} be quantifier-free formula that do not mention Br (but they can mention the maps in $\mathcal{G}$).

Then, if the following triple is valid:

$$\{\varphi \wedge \psi_{pre} \wedge Br = \emptyset\}\ P_{\mathcal{G},Br}\ \{\varphi \wedge \psi_{post} \wedge Br = \emptyset\}$$

then we must show that the following triple is valid:

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

where P is the projection of $P_{\mathcal{G},Br}$ to user-level code.

The core of the proof is the argument that for well-behaved programs, if

$$\{\varphi \wedge \psi_{pre} \wedge Br = \emptyset\}\ P_{\mathcal{G},Br}\ \{\varphi \wedge \psi_{post} \wedge Br = \emptyset\}$$

is valid then

$$\langle\, (\forall z \notin Br.\, \rho) \wedge \varphi \wedge \psi_{pre} \wedge Br = \emptyset\,\rangle\ P_{\mathcal{G},Br}\ \langle (\forall z \notin Br.\, \rho) \wedge \varphi \wedge \psi_{post} \wedge Br = \emptyset\,\rangle$$

is valid.

If we show that the latter is valid, then we can rewrite it by removing Br, showing the validity of the following triple:

$$\langle\, LC \wedge \varphi \wedge \psi_{pre}\,\rangle\ P_{\mathcal{G},Br}\ \ LC \wedge \varphi \wedge \psi_{post}\,\rangle$$

Observe that the specifications now do not mention Br. We then apply Proposition 3.4 to obtain the validity of the desired intrinsic triple. Therefore, we dedicate the rest of the section to showing the above property of well-behaved programs.

The proof proceeds by an induction on the nesting depth of method calls in a trace of the program $P_{\mathcal{G},Br}$. We elide this level of induction here because it is routine. Importantly, given a particular execution of the program P, we must show that the claim holds, assuming it holds for all method calls occurring in the execution. We show this by structural induction on the proof of well-behavedness of $P_{\mathcal{G},Br}$.

There are several base cases.

Skip/Assignment/Lookup/Return There is nothing to show for skip, assignment, lookup, or return statements. These do not change the heap at all and the rule does not update Br either, therefore if $\langle\, \alpha\,\rangle$ stmt $\langle\, \beta\,\rangle$ is valid then certainly $\langle\, (\forall z \notin Br.\, \rho) \wedge \alpha\,\rangle$ stmt $\langle\, (\forall z \notin Br.\, \rho) \wedge \beta\,\rangle$ is valid.

Mutation The claim is true for the mutation rule since by the premise of the rule we update the broken set with the impact set consisting of all potential objects where local conditions may not hold.

Function Call Here we simply appeal to the induction hypothesis.

Allocation We refer to our operational semantics, which ensures that no object points to a freshly allocated object. Therefore, the allocation of an object could have only broken the local conditions on itself at most.

Infer LC Outside Br There is nothing to prove for this rule as it does not alter the Br set at all.

Assert LC and Remove The claim holds for this rule by construction. If LC holds everywhere outside Br , and we know that $LC(x)$ holds, then we can conclude that LC holds everywhere outside $Br \setminus \{x\}$.

It only remains to show that the claim holds for larger well-behaved programs obtained by composing smaller well-behaved programs using sequencing, branching, or looping constructs. The proof here is trivial as the argument for sequencing is trivial (we can think of a loop as unboundedly many sequenced compositions of the smaller well-behaved program): we can *always* compose two well-behaved programs to obtain a well-behaved program.

B GHOST-ANNOTATED CODE FOR BST ROTATE

The following program for BST Rotate is based off of the local conditions and impact sets that appear in Section 4.2. Throughout the program, comments appear displaying the state of the broken set *Br* at that point.

```
pre: $Br = \emptyset \;\wedge\; l(x) \neq nil \;\wedge\; p(x) = xp$
post: $Br' = \emptyset \;\wedge\; p(ret) = xp$
      $\wedge\; l(ret) = old(l(l(x))) \;\wedge\; ret = old(l(x)) \;\wedge\; r(ret) = x$
      $\wedge\; l(r(ret)) = old(r(l(x))) \;\wedge\; r(r(ret)) = old(r(x))$
bst_right_rotate(x: BST, xp: BST?, Br: Set(BST))
returns ret: BST, Br: Set(BST)
{
  LCOutsideBr(x, Br);
  if (xp ≠ nil) then {
      LCOutsideBr(xp, Br);
  }
  if (x.l ≠ nil) then {
      LCOutsideBr(x.l, Br);
  }
  if (x.l ≠ nil ∧ x.l.r ≠ nil) then {
      LCOutsideBr(x.l.r, Br);
  }
  var y := x.l;                // {}
  Mut(x, l, y.r, Br);          // {x, y}
  if (xp ≠ nil) then {
      if (x = xp.l) then {
          Mut(xp, l, y, Br);  // {xp, x, y}
      }
      else {
          Mut(xp, r, y, Br);  // {xp, x, y}
      }
  }
  Mut(y, r, x, Br);            // {xp, x, y, x.l} (Note: x.l == old(y.r))
  // (1): Repairing x.l
  if (x.l ≠ nil) then {
      Mut(x.l, p, x, Br);     // {xp, x, y, x.l}
  }
  // (2): Repairing x
  Mut(x, p, y, Br);            // {xp, x, y, x.l}
  Mut(x, min, if x = nil then x.k else x.l.min, Br);    // {xp, x, y, x.l}
  // (3): Repairing y
  Mut(y, p, xp, Br);           // {xp, x, y, x.l}
  Mut(y, max, x.max, Br);      // {xp, x, y, x.l}
  Mut(y, rank, if xp = nil then x.rank+1 else (xp.rank+x.rank)/2, Br);     // {xp, x, y, x.l}
  AssertLCAndRemove(x.l, Br);  // {xp, x, y}
  AssertLCAndRemove(x, Br);    // {xp, y}
  AssertLCAndRemove(y, Br);    // {xp}
  AssertLCAndRemove(xp, Br);   // {}
  ret := y // return y
}
```

C LOCAL CONDITION, IMPACT SETS, AND GHOST-ANNOTATED CODE FOR SORTED LIST REVERSE

What follows are the complete local conditions and impact sets for Sorted List Reverse:

$$
\begin{aligned}
LC \equiv \forall x.\ & prev(x) \neq nil \Rightarrow next(prev(x)) = x \\
& \land\ next(x) \neq nil \Rightarrow prev(next(x)) = x \\
& \qquad \land\ length(x) = length(next(x)) + 1 \\
& \qquad \land\ keys(x) = keys(next(x)) \cup \{key(x)\} \\
& \qquad \land\ hslist(x) = hslist(next(x)) \uplus \{x\} \\
& \qquad \land\ sorted(x) \Rightarrow key(x) \leq key(next(x)) \\
& \qquad\qquad \land\ sorted(x) = sorted(next(x)) \\
& \qquad \land\ rev\_sorted(x) \Rightarrow key(x) \geq key(next(x)) \\
& \qquad\qquad \land\ rev\_sorted(x) = rev\_sorted(next(x)) \\
& \land\ (next(x) = nil \Rightarrow length(x) = 1 \land keys(x) = \{x\} \land hslist(x) = \{x\})
\end{aligned}
$$

Fig. 9. Full local condition for lists for Sorted List Reverse

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

Table 3. Full impact sets for lists for Sorted List Reverse

The following program reverses a sorted list as defined by the local condition above. Like in Appendix B, we annotate this program with comments on the current composition of the broken set according to the rules of Table 3.

```
pre: $Br = \emptyset \ \land\ \varphi(x) \ \land\ sorted(x)$
post: $Br' = \emptyset \ \land\ \varphi(ret) \ \land\ rev\_sorted(ret) \ \land\ keys(ret) = old(keys(x)) \ \land\ hslist(ret) = old(hslist(x))$
sorted_list_reverse(x: C, Br: Set(C))
returns ret: C, Br: Set(C)
{
  LCOutsideBr(x, Br);
  var cur := x;
  ret := null;
  while (cur ≠ nil)
     invariant $cur \neq nil \ \Rightarrow\ LC(cur) \land sorted(cur) \land \varphi(cur)$
     invariant $ret \neq nil \ \Rightarrow\ LC(ret) \land rev\_sorted(ret) \land \varphi(ret)$
     invariant $cur \neq nil \ \land\ ret \neq nil \ \Rightarrow\ key(ret) \ \leq\ key(cur)$
     invariant $old(keys(x)) = ite(cur = nil,\ \emptyset,\ keys(cur)) \cup ite(ret = nil,\ \emptyset,\ keys(ret))$
```

```
      invariant old(hslist(x)) = ite(cur = nil,  ∅,  hslist(cur)) ∪ ite(ret = nil,  ∅,  hslist(ret))
      invariant Br = ∅
      decreases ite(cur ≠ nil,  0,  length(cur))
   {
      var tmp := cur.next;                 // {}
      if (tmp ≠ nil) then {
        LCOutsideBr(tmp, Br);              // {}
        Mut(tmp, p, nil, Br);              // {cur, tmp}
      }
      Mut(cur, next, ret, Br);             // {cur, tmp}
      if (ret ≠ nil) then {
        Mut(ret, p, cur, Br);              // {cur, tmp, ret}
      }
      Mut(cur, keys,
        {cur.k} ∪ (if cur.next=nil then φ else cur.next.keys), Br);    // {cur, tmp, ret}
      Mut(cur, hslist,
        {cur} ∪ (if cur.next=nil then φ else cur.next.hslist), Br);    // {cur, tmp, ret}
      if (cur.next ≠ nil ∧ (cur.key > cur.next.key ∨ ¬cur.next.sorted)) {
        Mut(cur, sorted, false, Br);       // {cur, tmp, ret}
      }
      Mut(cur, rev_sorted, true, Br);      // {cur, tmp, ret}
      AssertLCAndRemove(cur, Br);          // {tmp, ret}
      AssertLCAndRemove(ret, Br);          // {tmp}
      AssertLCAndRemove(tmp, Br);          // {}
      ret := cur;
      cur := tmp;
   }
   // The current value of ret is returned
 }
```

D LOCAL CONDITION AND IMPACT SETS FOR SORTED LIST MERGE

Here is the local condition for disjoint sorted lists, which combine sorted list conditions with monadic maps $list1, list2, list3$ that ensure that lists of a particular class (represented by these monadic maps) are disjoint from lists of another class.

$$\begin{aligned}
LC \equiv \forall x.(&list1(x) \vee list2(x) \vee list3(x)) \\
&\wedge\ \neg(list1(x) \wedge list2(x)) \wedge \neg(list2(x) \wedge list3(x)) \\
&\wedge\ \neg(list1(x) \wedge list3(x)) \\
&\wedge\ (prev(x) \neq nil \Rightarrow next(prev(x)) = x) \\
&\wedge\ (next(x) \neq nil \Rightarrow prev(next(x)) = x \\
&\qquad\qquad \wedge\ length(x) = length(next(x)) + 1 \\
&\qquad\qquad \wedge\ keys(x) = keys(next(x)) \cup \{key(x)\} \\
&\qquad\qquad \wedge\ hslist(x) = hslist(next(x)) \uplus \{x\} \quad \text{(disjoint union)} \\
&\qquad\qquad \wedge\ key(x) \leq key(next(x))) \\
&\wedge\ (next(x) = nil \Rightarrow length(x) = 1 \wedge keys(x) = \{key(x)\} \wedge hslist(x) = \{x\}) \\
&\wedge\ (list1(x) \Rightarrow (next(x) \neq nil \Rightarrow list1(next(x)))) \\
&\wedge\ (list2(x) \Rightarrow (next(x) \neq nil \Rightarrow list2(next(x)))) \\
&\wedge\ (list3(x) \Rightarrow (next(x) \neq nil \Rightarrow list3(next(x))))
\end{aligned} \tag{3}$$

Fig. 10. Full local condition for lists for Sorted List Reverse

Note that we also have a variation of the local condition LC_{NC}, used in ghost loop invariants, which is similar to Equation 3, except the final three conjuncts (those enforcing closure on $list1, list2, list3$) are removed. This is done when converting an entire list from one class to another (i.e., converting from $list1$ to $list3$). The following are the full impact sets for all fields of this data structure.

Mutated Field f	Impacted Objects A_f
$next$	$\{x, old(next(x))\}$
key	$\{x, prev(x)\}$
$prev$	$\{x, old(prev(x))\}$
$length$	$\{x, prev(x)\}$
$keys$	$\{x, prev(x)\}$
$hslist$	$\{x, prev(x)\}$
$list1$	$\{x, prev(x)\}$
$list2$	$\{x, prev(x)\}$
$list3$	$\{x, prev(x)\}$

Fig. 11. Full impact sets for disjoint sorted lists