Generalized Arrays for Stainless Frames

Georg Stefan Schmid* and Viktor Kunčak

LARA, EPFL, Switzerland
{georg.schmid, viktor.kuncak}@epfl.ch

Abstract. We present an approach for verification of programs with shared mutable references against specifications such as assertions, preconditions, postconditions, and read/write effects. We implement our tool in the Stainless verification system for Scala.
A novelty of our approach is to translate imperative function contracts (including frame conditions) using quantifier-free formulas in first-order logic, instead of quantifiers or separation logic. Our quantifier-free encoding enables SMT solvers to both prove safety and to report counterexamples relative to the semantics of procedure contracts. Our encoding is possible thanks to the expressive power of the extended array theory of de Moura and Bjørner, implemented in the SMT solver Z3, whose map operators allow us to project heaps before and after the call onto the declared reads and modifies clauses.
To support inductive proofs about the preservation of invariants, our approach permits capturing a projection of heap state as a history variable and evaluating imperative ghost code in the specified captured heap.
We also retain the efficiency of reasoning about purely functional layers of data structures, which need not be represented using heap references but often map directly to SMT-LIB algebraic data types and arrays. We thus obtain a combination of expressiveness for shared mutable data where needed, while retaining automation for purely functional program aspects. We illustrate our approach by proving detailed correctness properties of examples manipulating mutable linked structures.

Keywords: verification · satisfiability modulo theories · shared mutable data structures · array theories · dynamic frames

1 Introduction

Formal verification of programs with shared mutable data structures is a long-standing problem. Among the most promising techniques used in today's verification tools are separation logic and dynamic frames. Separation logic [33] with bi-abduction [9] has proved practical; its variant is implemented in the Infer tool [12] used by Facebook. It is also a common framework for foundational semantic-based approaches for reasoning about state inside the Coq proof assistant [20]. On the other hand, we are attracted to dynamic frames [21] because they are both semantically straightforward and expressive. Tools that embrace

* Corresponding author.

them, such as Dafny [25], were used to verify complex software systems at Microsoft [18]. Separation logic and dynamic frames are closely related and one can view separation logic as a logical framework that infers sets that represent dynamic frames in certain circumstances, as illustrated by the VeriFast tool [37], a relationship that was rigorously analyzed in subsequent research [34].

This paper presents an alternative approach for reasoning about mutable programs and presents its realization in the Stainless verifier [17] for a subset of the Scala programming language [32]. Like the dynamic frames approach, we use constrained sets of objects to specify frame conditions. Like Dafny, our tool uses SMT solvers to establish properties instead of dedicated symbolic execution for heap-manipulating programs as in several other approaches [5,13,19,30]. We also model the heap as a function from storage locations to values.

However, our encoding of frame conditions is different from the one in Dafny. Whereas Dafny makes use of *universal quantifiers with triggers* to encode frame conditions (expressing that *all* non-modified locations remain the same), we avoid quantifiers and instead use the *generalized theory of arrays* [29] of Z3. Notably, this expressive array theory retains completeness guarantees for satisfiability checking of quantifier-free formulas even in the presence of model-based theory combination [27] with other decidable theories. Thanks to our new encoding and the decision procedures of Z3, our verification tool can report meaningful counterexamples for invalid properties, even in those cases where the bodies of methods are abstracted by their modifies clauses. In contrast, SMT solvers either refuse to report counterexamples to satisfiability for formulas with universal quantifiers, or permit extraction of assignments that may or may not be witnesses to satisfiability. Unlike Dafny, which reduces programs to a guarded-command language Boogie [3], our approach reduces imperative code to recursive functional programs that manipulate data types supported by the Z3 SMT solver [28], building on the existing Stainless infrastructure [17]. While Stainless could already deal with imperative constructs [7], the supported fragment did not permit any aliasing. In contrast, the new encoding we describe enables Stainless to verify shared mutable data structures.

Our approach reduces verification conditions to functional programs but need not encode immutable algebraic data types using the heap. Read-only functions do not return a heap in our encoding, whereas functions that do not read mutable references do not even take a heap argument. The result is a better verification experience on a mix of purely functional and mutable code, compared to a more uniform encoding. This feature enables users to leverage the expressive power of recursive functional programming in implementation and specification, and encourages the use of executable specifications. Following this paradigm, we further allow users to define inductive heap predicates as Boolean-typed recursive functions. Lemmas about such predicates typically require inductive proofs and the ability to explicitly relate to states at different program points. We propose first-class heaps as a solution which provides the necessary fine-grained control and is readily expressible using our approach.

Contributions. This paper makes the following contributions:

- We describe a novel translation of frame conditions into quantifier-free formulas of combinatory array logic, yielding a heap encoding that can reliably produce abstract counterexamples modulo function contracts.
- We show how to soundly incorporate into our approach the notion of first-class heaps, affording additional flexibility in proving lemmas about inductive heap predicates, while coming at essentially no additional cost in translation. First-class heaps also increase our system's expressive power in that they enable writing proofs of hyperproperties [10, 16, 22].
- We integrate our solution into the Stainless verifier. Our implementation supports imperative and functional features, including higher-order functions and generics, and uses dynamic frames as a specification mechanism.[1]

2 First Example: Stack

As a simplest example to illustrate a mix of functional and imperative programming, Figure 1 presents a mutable stack implementation using the textbook singly-linked list. (The code is valid Scala accepted by the Scala 2.12/2.13 compilation pipeline given appropriate library imports.) The data structure is simple to specify: a minimal specification would only include reads and modifies clauses, with bodies of functions themselves serving as specifications.

Figure 1 extends such basic specification by introducing the abstraction function list and calling it in postconditions (**ensuring**) to re-state the precise effect of the function. For instance, the postcondition of push states that list == a :: old(list), meaning that the result of invoking parameter-less abstraction function list in the post-state is structurally equivalent (==) to element a cons-ed (::) with list evaluated in the pre-state (old(list)). The proofs of all these conditions in push and pop are trivial and our system performs them in a fraction of a second. The clients can reason about the behavior of stack by referring to the immutable list, which is suited for inductive proofs, much like such list data types in proof assistants Coq [6] and Isabelle [31]. Users can create shared references to such mutable stacks, which goes beyond what was possible with the previous, unique mutable reference model of Stainless, inherited from Leon [7, Ch. 3].

3 Extended Example: Map on a Tree

Moving to a slightly more complex example, Figure 2 shows a binary tree data whose interior nodes are immutable but whose leaves are mutable and store values of generic type T. We support a fragment of Scala with functional features (such as pure first-class functions) as well as imperative features (mutable fields)

[1] Our implementation is part of Stainless (https://github.com/epfl-lara/stainless/) and can be tested on examples in `frontends/benchmarks/full-imperative` via the `--full-imperative` flag. Artifact available at https://zenodo.org/record/5683321.

```
case class Stack[T](private var data: List[T]) extends AnyHeapRef {
  def list = {
    reads(Set(this))
    data
  }

  def push(a: T): Unit = {
    reads(Set(this))
    modifies(Set(this))
    data = a :: data // executable code
  } ensuring(_ ⇒ list == a :: old(list))

  def pop: T = {
    reads(Set(this))
    require(!list.isEmpty)
    modifies(Set(this))
    val n = data.head // executable code
    data = data.tail // executable code
    n // executable code
  } ensuring (res ⇒ res == old(list).head && list == old(list).tail)
}
```

Fig. 1. A mutable stack.

```
case class Cell[T](var value: T) extends AnyHeapRef

case class Leaf[T](data: Cell[T]) extends Tree[T]
case class Branch[T](left: Tree[T], right: Tree[T]) extends Tree[T]
sealed abstract class Tree[T] {
  @ghost def repr: Set[AnyHeapRef] = this match { // all cells in the tree
    case Leaf(data) ⇒ Set[AnyHeapRef](data)
    case Branch(left, right) ⇒ left.repr ++ right.repr
  }

  def tmap(f: T ⇒ T): Unit = { // minimal specification
    reads(repr)
    modifies(repr)

    this match {
      case Leaf(data) ⇒ data.value = f(data.value)
      case Branch(left, right) ⇒ left.tmap(f); right.tmap(f)
    }
  }
}
```

Fig. 2. A tree with mutable leaves and a parallelizable in-place map, including read and write frame conditions. The ++ symbol denotes union of sets, as in Scala.

and object-oriented features (traits and dynamic dispatch). For any class, users explicitly opt into mutability and heap reasoning by inheriting from AnyHeapRef. For instance, in our example the class Tree inherits from AnyHeapRef. It is also marked as **sealed**, indicating that all of Tree's subclasses are defined locally (as opposed to Scala's default behavior of keeping type hierarchies *open*). In effect, Tree constitutes an algebraic data type with constructors Leaf and Branch.

Our focus is the method **def** tmap(f: T ⇒ T) on the Tree class, which applies an in-place transformation f to all leaf cells. For example, given a tree: Tree[BigInt], invoking tree.tmap(n ⇒ n + 1) increments the values in all the leaves of tree by one. The method recursively traverses the tree and updates all cells upon reaching the leaves.

Verifying Effects. Figure 2 is also a minimally-specified program accepted by our tool, which automatically verifies the conformance of tmap to its declared effects. The **reads** clause indicates that the only mutable references that tmap reads are given by the value returned from auxiliary function repr, which computes the set of mutable cells in a given tree. Similarly, **modifies** indicates that these are the only sets the method is allowed to modify, which means that all other mutable objects remain the same after a call to tmap. The **@ghost** annotation ensures that the repr function is not accidentally executed, but can only be used in specifications that are erased at run time.

If we try to omit a reads or modifies clause, or incorrectly define repr to not descend into subtrees, the tool reports a counterexample state detecting that the specification **reads** or **modifies** is violated, with a message such as

tmap body assertion: reads of Tree.tmap **invalid**

pointing to an undeclared effect in line 17 of Figure 2.

Counterexamples. Our approach enables the generation of counterexamples on the basis of function contracts alone. Consider the following test method:

```
def test[T](t: Tree[T], c: Cell[T], y: T) = {
  reads(t.repr ++ Set[AnyHeapRef](c))
  modifies(t.repr)

  t.tmap(x ⇒ y)
} ensuring(_ ⇒ c.value == old(c.value))
```

If we mark tmap using the **@opaque** annotation to prevent it from being unfolded and try to verify test, the system reports a counterexample, such as this one:

```
Found counter−example:
  t: Tree[T] → Leaf[Object](HeapRef(12))
  c: HeapRef → HeapRef(12)
  y: T → SignedBitvector32(1)
  heap0: Map[HeapRef, Object] → {HeapRef(12) →
       Cell(Cell[Object](SignedBitvector32(0))), ∗ → SignedBitvector32(2)}
```

indicating that, when tmap is approximated with its effects, the **ensuring** clause can be violated when tree t contains precisely the reference c.

Tools such as Dafny have difficulties in discovering such counterexamples, as they rely on an encoding of frame conditions that involves quantifiers. Aiming for soundness of counterexamples, the underlying SMT solvers may refuse to produce any output or, in some cases, may produce an assignment that is not guaranteed to be a model. This limitation is due to the fact that certifying that a model exists in the presence of general quantifiers is a very difficult problem. Generalized arrays [29] avoid it by "building in" restricted forms of quantifiers into the semantics of pointwise (map) operators, improving the predictability.

Verifying Functional Correctness. To illustrate specification of stronger correctness properties, we show that tmap behaves like map on purely functional lists. This stronger specification of tmap is in the **ensuring** (postcondition) clause of the version of tmap in Figure 3 (line 18). The property is interesting because it gives us assurance of correctness while being able to write code that reuses memory locations and permits parallelization. The property is expressed by defining an abstraction function [1] toList that maps the tree into the sequence of elements stored in its leaf cells. (The purely functional List data type and the map function on lists are defined in the standard library of Stainless.) To prove the **ensuring** clause, it is necessary to introduce a precondition for tmap, expressed using the construct **require**(valid). The valid method returns true when all subtrees store disjoint cells. The tmap method may then only be called when this predicate holds. The assertion on line 14 follows directly from valid and expresses disjointness of the side effects of calls on line 15.

In many cases our tool can automatically prove properties of interest thanks to SMT solvers and the unfolding algorithm of Stainless. For instance, the valid method (which we use to establish separation of subtrees) does not depend on the content of mutable cells, but only on the identity of references. Our tool checks this independence thanks to the absence of **reads** and **modifies** clauses in the signature of valid. Because it does not depend on mutable state, valid trivially continues to hold after each invocation of tmap on line 15.

On the other hand, showing complex properties such as functional correctness may require more elaborate reasoning. The first challenge in our example is to establish on line 16, *after* the modifications have taken place, the correctness property we desire for each subtree, i.e., left.toList == oldList1.map(f) and right.toList == oldList2.map(f). This requires using the heap separation between left and right (witnessed by valid) to deduce that the two recursive calls are in fact entirely independent of another. This, in turn, requires taking into account tmap's **modifies** clause, which states that only objects in repr are modified. In previous works such a clause is encoded in one of two ways. Systems such as Dafny encode frame axioms as quantified first-order formulas and rely on *triggers* to automate their instantiation. In contrast, separation logic verifiers explicitly control the choice of frame, and thus move the burden of instantiations out of the SMT solver. We propose a third solution, which is to encode the frame conditions as quantifier-free assumptions in array theory, injected at each

```
1  def tmap(f: T ⇒ T): Unit = { // strong specification
2    reads(repr)
3    modifies(repr)
4    require(valid)
5    @ghost val oldList = toList
6
7    this match {
8      case Leaf(data) ⇒
9        data.value = f(data.value)
10       ghost { check(toList == oldList.map(f)) }
11
12     case Branch(left, right) ⇒
13       @ghost val (oldList1, oldList2) = (left.toList, right.toList)
14       assert(left.repr ∩ right.repr == ∅)
15       left.tmap(f); right.tmap(f)
16       ghost { lemmaMapConcat(oldList1, oldList2, f) }; ()
17   }
18 } ensuring (_ ⇒ toList == old(toList.map(f))) // main property
19
20 def valid: Boolean = // tree invariant: subtrees store disjoint cells
21   this match {
22     case Leaf(data) ⇒ true
23     case Branch(left, right) ⇒
24       left.repr ∩ right.repr == ∅ &&
25       left.valid && right.valid
26   }
27
28 def toList: List[T] = { // abstraction function
29   reads(repr)
30   this match {
31     case Leaf(data) ⇒ List(data.value)
32     case Branch(left, right) ⇒ left.toList ++ right.toList
33   }
34 }
35
36 def lemmaMapConcat[T, R](xs: List[T], ys: List[T], f: T ⇒ R): Unit = {
37   xs match {
38     case Nil() ⇒ ()
39     case Cons(_, xs) ⇒ lemmaMapConcat(xs, ys, f)
40   }
41 } ensuring (_ ⇒ xs.map(f) ++ ys.map(f) == (xs ++ ys).map(f))
```

Fig. 3. Functional correctness of the tmap method including the abstraction function, the invariant, and a proven lemma about purely functional lists. We use ∩ to display intersection of sets, and use ∅ for the empty set of heap references `Set[AnyHeapRef]()`. The ++ symbol denotes concatenation of functional lists and union of sets, as in Scala.

function call site. Our approach avoids the need for quantifiers, but retains the automation of SMT solvers.

Despite that automation and the decidability of the generalized array theory, the size and complexity of SMT formulas may overwhelm the solver. In such cases the user can add auxiliary assertions, e.g., expressed through **assert** and **check** statements in Figure 3. Furthermore, certain properties may require explicit guidance on inductive proofs when reasoning does not follow the pattern of functions that are iteratively unfolded. In such cases, we need to introduce lemmas and prove them using recursion to express inductive arguments, as with lemmaMapConcat defined in lines 36-41 and instantiated on line 16. This lemmas is independent of any state reasoning and would naturally fit in a standard list library. With these specifications and hints in place, our tool successfully verifies the functional correctness of tmap.

4 First-Class Heaps

For some proofs it is useful to directly refer to and manipulate the heap states at different points in the program. In our system's surface language we expose heaps as first-class values of abstract type Heap, and our standard library contains several primitives to manipulate such values: a function Heap.get which returns the current implicit heap, a primitive h.eval(e) which evaluates expression e in the context of heap h, and the function Heap.unchanged(s, h0, h1) which evaluates to true iff there exists no object o in the set s: Set[AnyHeapRef] such that heaps h0 and h1 interpret o differently (in the shallow sense).

For instance, we might want to re-establish an inductive heap predicate after having modified a node-based data structure:

```
case class Node(var next: Option[Node]) extends AnyHeapRef

def sll(nodes: List[Node]): Boolean = {
  reads(nodes.content.asRefs)
  nodes match {
    case Cons(node1, rest @ Cons(node2, _)) ⇒
      node1.next == Some(node2) && sll(rest)
    case _ ⇒ true
  } }
```

In the above example we have a heap type of Nodes with pointers to next nodes and an inductive heap predicate, sll, witnessing that a given sequence of nodes forms a singly-linked list. Note that nodes: List[Node] itself is a purely functional data structure and only present for specification purposes; one would typically store it as a **@ghost** variable.

Say we would like to prove that removing the last element of a non-empty singly-linked list nodes maintains the sll property. This is easy to specify using our functional abstraction nodes: assuming sll(nodes) holds in the pre-state, we would like to show that sll(nodes.init) holds in the post-state, where

.init is a method in the standard library that drops the last element of a List[T]. When nodes consists of a single element, the property follows immediately, since sll(nodes.init) reduces to sll(Nil) which holds by definition of sll. On the other hand, if nodes contains at least two elements, we need to modify the next field of the second-to-last node, i.e., set nodes(nodes.size − 2).next = None(). In the latter case we effectively want to establish the Hoare triple

$$\{\text{sll(nodes)} \wedge \text{F}\}\ \text{nodes(nodes.size} - 2\text{).next} = \text{None()}\ \{\text{sll(nodes.init)}\}$$

where F is some additional precondition ensuring that the list has at least two elements, and that all nodes up to the last two are separate from the rest.

```
// A lemma proving that popping from a SLL maintains singly−linked−ness.
def sllPopLemma(h0: Heap, h1: Heap, nodes: List[Node]): Unit = {
  require(
    nodes.nonEmpty &&
    h0.eval { sll(nodes) } &&
    (nodes.size == 1 || (
      Heap.unchanged(nodes.init.init.content.asRefs, h0, h1) &&
      h1.eval { nodes(nodes.size − 2).next == None() }
    )) )
  if (nodes.size > 1) sllPopLemma(h0, h1, nodes.tail)
} ensuring (_ ⇒ h1.eval { sll(nodes.init) })
```

Above, sllPopLemma establishes the desired property by explicitly referring to the pre-state as h0 and the post-state as h1. Its proof proceeds by induction on nodes, and is mostly automatic; we merely have to invoke the right induction hypothesis when nodes.size > 1. An implementation of pop would likely resort to a stronger invariant like distinctness of all objects in nodes, and then invoke the lemma after the modification as follows

```
val h0 = Heap.get // Get the pre−state
if (nodes.size > 1) nodes(nodes.size − 2).next = None() // Unlink last element
sllPopLemma(h0, Heap.get, nodes)
```

along with some hints that deduce F from the stronger invariant (not shown). In addition, for nodes to be marked **@ghost**, we would need to maintain nodes(nodes.size − 2) in a separate non-**@ghost** variable. Our benchmark suite includes similar, but more elaborate examples Queue and NodeCycle.

While our current system does not provide as much automation as separation logic for tree-like data, our approach is not limited to such structures and retains full flexibility in treating heaps as first-class values. Interestingly, this also enables us to prove hyperproperties, i.e., properties such as determinism, which involve multiple heap states. For example, consider the following lemma stating that a memoized function f : Int ⇒ Int evaluates to the same result in every heap:

```
def lemmaHeapIsIrrelevant(h0: Heap, h1: Heap, x: Int) = { () }
  ensuring (_ ⇒ h0.eval { f(x) } == h1.eval { f(x) })
```

In many cases such lemmas can be proven automatically by our system, as demonstrated, for instance, by the FibCache benchmark.

5 Heap Encoding

In the following, we introduce our heap encoding and how it achieves framing without quantification. Our approach builds upon the existing counterexample-complete unfolding procedure of the Stainless verifier and exploits the additional expressive power afforded by combinatory array logic [29], an extended array theory available in Z3. This use of array combinators for framing is, to the best of our knowledge, novel. Notably, our encoding allows for a high degree of proof automation without giving up counterexamples.

Our tool models stateful operations by explicitly reading from and updating a locally-mutable map that relates each object to its state. In a later transformation step such programs with local mutations are reduced to functional ones. Each stateful function gains an explicit heap parameter and returns a new, potentially updated heap along with its regular output. In terms of Scala's type system, the heap can be thought of as a map heap of type HeapMap = Map[HeapRef, Any] where Any is the top type and HeapRef is a data type representing an object's identity. Conceptually, our approach employs a monadic translation [26, 41] that we partially-evaluate [2], replacing stateful operations such as reads and writes by pure operations on a map.

5.1 Encoding tmap

We first give an informal explanation of our encoding by the example of the minimally-specified version of tmap on Tree (the version without postconditions, shown in Figure 2). In Figure 4 we show the data types after transformation.

We treat *heap types*, i.e., descendants of AnyHeapRef, like Cell, differently from immutable types such as Tree. The latter are translated into algebraic data types in the obvious way (lines 5-7). References to heap types, on the other hand, are erased to the internal ADT HeapRef that represents locations on the heap (line 1). For instance, the field data: Cell[T] of Leaf becomes $\mathsf{data}_{\mathsf{ref}}$: HeapRef (line 6). Additionally, each heap class like Cell is translated to a single-constructor ADT that encapsulates an object's state at a given time, e.g., $\mathsf{Cell}_{\mathsf{Data}}$ (line 3).

In Figure 5 we show the encoding of tmap itself. The method is reduced to a type-parametric function that takes its original argument f, the method receiver t and a heap parameter h0. The imperative operations in tmap are translated to functional operations on HeapMap as mentioned above, and the modified heap is returned along with the original return value. In particular, if the current tree t is a leaf, then we extract its reference to a cell $\mathsf{data}_{\mathsf{ref}}$ (line 4) and index the initial heap h0 at $\mathsf{data}_{\mathsf{ref}}$ (line 9). Note that since the heap map stores values of type Any we have to perform a downcast (lines 8-9). This is safe, since we will only verify well-typed Scala programs, so any such cast will be correct by construction. In a later type-encoding phase [40] Stainless translates type tests such as line 8 to conditions in the theory of inductive data types. On line 11 we apply the function f to the old value of data and construct a $\mathsf{Cell}_{\mathsf{Data}}$ value reflecting the new state of data. We then return the updated heap on line 12. In

```
case class HeapRef(id: BigInt)

case class Cell_Data[T](value: T)

sealed abstract class Tree[T]
case class Leaf[T](data_ref: HeapRef) extends Tree[T]
case class Branch[T](left: Tree[T], right: Tree[T]) extends Tree[T]
```

Fig. 4. The data types of the tmap example in Figure 2 after our encoding.

```
def tmap[T](h0: HeapMap, t: Tree[T], f: T ⇒ T): (Unit, HeapMap) = {
  val (rs, ms) = (repr(t), repr(t))
  t match {
    case Leaf(data_ref) ⇒
      assert(data_ref ∈ rs, "'data' must be in reads set")
      assert(data_ref ∈ ms, "'data' must be in modifies set")
      val data: Cell_Data = {
        assume(h0(data_ref).isInstanceOf[Cell_Data[T]])
        h0(data_ref).asInstanceOf[Cell_Data[T]]
      }
      val data': Cell_Data = Cell_Data(f(data.value))
      ((), h0.updated(data_ref, data'))

    case Branch(left, right) ⇒
      val (_, h1) = tmap_shim(h0, rs, ms, left, f)
      tmap_shim(h1, rs, ms, right, f)
  }
}

def tmap_shim[T](h0: HeapMap, rd: RSet, md: RSet, t: Tree[T], f: T ⇒ T): (Unit,
    HeapMap) = {
  val (rs, ms) = (repr(t), repr(t))
  assert(rs ⊆ rd, "reads set of Tree.tmap")
  assert(ms ⊆ md, "modifies set of Tree.tmap")
  val res = tmap(h0, t, f)
  val resR = tmap(rs.mapMerge(h0, dummyHeap), t, f)
  assume(res._1 == resR._1)
  assume(res._2 == ms.mapMerge(resR._2, res._2))
  assume(res._2 == ms.mapMerge(res._2, h0))
  res
}
```

Fig. 5. The result of encoding the minimally-specified tmap method of Figure 2. We use ⊆ to typeset subsetOf, ∈ for contains, and abbreviate Set[AnyHeapRef] by RSet.

case the tree t is a Branch we simply perform two recursive calls (lines 15-16), albeit through the newly-introduced wrapper function $\mathsf{tmap}_{\mathsf{shim}}$.

Our encoding achieves modular verification of heap contracts (**reads** and **modifies**) by injecting some additional assertions and assumptions. We bind the **reads** and **modifies** sets (rs and ms) at the top of the function (line 2). For each object that is read or modified we check that the object is in the respective set (lines 5-6). For function calls we check that the callee's **reads**, resp. **modifies**, set is subsumed by the caller's. We achieve this by invoking a wrapper function $\mathsf{tmap}_{\mathsf{shim}}$, that additionally takes as parameters the *domains* on which the passed heap is defined for reads and modifications (rd and md). Within the wrapper we bind the original function's **reads** and **modifies** sets (line 21), check subsumption wrt. the domains (lines 22-23) and call the original function tmap (line 24).

Finally, we assume the modular guarantees about tmap wrt. the pre- and post-state, i.e., its *frame conditions*: On lines 26-27 we state that the result of tmap only depends on the **reads** subset of the heap, whereas on line 28 we state that the heap resulting from tmap may only have changed on objects in **modifies**. For the **reads**-related frame conditions we depend on a "hypothetical" application of f to the projected heap rs.mapMerge(h0, dummyHeap), which contains the state of h0 for all objects in rs and that of dummyHeap elsewhere. The first assumption thus states that the result computed by f is the same no matter whether we apply it to h0 or to some other arbitrary (but well-typed) heap that is only known to agree on the valuations of objects in rs. The second assumption states the analogous property about the locations that might have been modified by f. Finally, the third assumption expresses that the pre-state equals the post-state in all locations but those in the **modifies** clause, i.e., the set ms.

The crucial component of our encoding here is the *mapMerge primitive*, which can be seen as a ternary operator of type $\forall$ K V. Set[K] $\Rightarrow$ Map[K,V] $\Rightarrow$ Map[K,V] $\Rightarrow$ Map[K,V]. Specifically, mapMerge takes a set s along with two maps m1, m2 and produces a map m' = s.mapMerge(m1, m2) such that $\forall$ k:K. (k $\in$ s $\rightarrow$ m'[k] = m1[k]) $\wedge$ (k $\notin$ s $\rightarrow$ m'[k] = m2[k]). We will discuss how mapMerge is translated to Z3's extended array theory in Section 5.3.

5.2 Translation Rules

We now describe the general translation rules as applied in our system. We will consider only a subset of the language supported, focussing on constructs of particular interest in the translation (shown in Figure 6).

We distinguish the terms t and types T of the surface language from those of the language after encoding. The surface language comprises of both (immutable) algebraic data types D and (mutable) heap types C, along with terms for field reads t.f and updates t.f := t, which are interpreted as either functional or imperative operations, depending on whether the receiver is an ADT or a heap type. In the lowered language the latter are always interpreted functionally, and the only imperative feature available are locally-mutable variables **let var** x = t **in** t and assignments thereof, x := t. Though not discussed here, it is straightforward

Variables ... x, y, h, ρ, μ

Surface Language

$$
\begin{array}{rrl}
\text{Types} \ldots & S, T & := \quad C \mid D \mid \textbf{Set}[T] \mid \textbf{AnyHeapRef} \\
\text{Terms} \ldots & t & := \quad x \mid f(\overline{t}) \mid \textbf{let}\ x = t\ \textbf{in}\ t \mid t.f \mid t.f := t \\
\text{Functions} \ldots & f & := \quad \textbf{def}\ f(\overline{x : T}) : S = \{\textbf{reads}(t);\ \textbf{modifies}(t);\ t\}
\end{array}
$$

Lowered Language

$$
\begin{array}{rrl}
\text{Types} \ldots & S, T & := \quad D \mid \textbf{Set}[T] \mid \textbf{Map}[T, T] \mid \textbf{Any} \mid \textbf{HeapRef} \\
\text{Terms} \ldots & t & := \quad x \mid f(\overline{t}) \mid \textbf{let}\ x = t\ \textbf{in}\ t \mid t.f \mid t.f := t \mid \\
 & & \quad\ \ \textbf{let var}\ x = t\ \textbf{in}\ t \mid x := t \mid \\
 & & \quad\ \ t\,[t] \mid t.\mathsf{update}(t, t) \mid t.\mathsf{mapMerge}(t, t) \mid \\
 & & \quad\ \ t.\mathsf{isInstOf}[T] \mid t.\mathsf{asInstOf}[T] \mid \\
 & & \quad\ \ \textbf{assume}(t); t \mid \textbf{assert}(t); t \\
\text{Functions} \ldots & f & := \quad \textbf{def}\ f(\overline{x : T}) : S = \{t\}
\end{array}
$$

Fig. 6. Selected terms and types of the languages before and after heap encoding.

to convert programs with local mutation into purely functional ones [7,15]. Our simplified language also omits first-class functions. In practice, we require them to be pure, while side-effectful ones can be encoded using abstract classes with heap contracts (see Task in Figure 10 for an example).

At its heart, our translation turns imperative operations on heap types $C_1, C_2, \ldots$ into functional operations on a map representing the entire heap. What should be the key and value types of the heap map? For keys, i.e., the references in our heap model, we choose an abstract type **HeapRef** isomorphic to the natural numbers, but with equality as its only operation. For values, i.e., the state of individual objects, we pick the top type **Any** as the trivial solution which subsumes the representations of all heap types. While SMT solvers do not directly support subtyping, this is convenient in Stainless, as we can leverage its existing support for subtyping and **Any** [40]. Our design differs from that supported by the Boogie verifier, whose type system provides higher-rank map types [24] in which the heap map may be typed as ∀T. Map[Ref[T], T], avoiding the need for (correct-by-construction) downcasts and an additional type encoding phase to deal with the Any type.

Due to our choice of heap representation, the lowered language includes maps and type-tests to express various assumptions about the heap that are correct by construction. For maps, we use $t\,[t_k]$ to denote indexing and $t.\mathsf{update}(t_k, t_v)$ to denote the (functional) result of updating a map t at key t_k. To recover information from **Any**-typed values, we provide $t.\mathsf{isInstOf}[T]$ to express type tests

and $\mathsf{t}.\mathsf{asInstOf}[\mathsf{T}]$ for the corresponding downcasts. Furthermore, $\mathbf{assume}(\mathsf{t});\mathsf{t}$ and $\mathbf{assert}(\mathsf{t});\mathsf{t}$ mark assumptions and assertions to be used during VC generation. Combining these constructs, we can express a downcast of t to T that is assumed correct as $\mathbf{let}\ \mathsf{x} = \mathsf{t}_1\ \mathbf{in}\ \mathbf{assume}(\mathsf{x}.\mathsf{isInstOf}[\mathsf{T}]);\mathsf{t}_2\{\mathsf{x} \mapsto \mathsf{x}.\mathsf{asInstOf}[\mathsf{T}]\}$, which we abbreviate by $\mathbf{let}\ \mathsf{x} = \mathsf{t}_1\ \mathbf{as}\ \mathsf{T}\ \mathbf{in}\ \mathsf{t}_2$. As in the example in Section 5.1, we take **HeapMap** and **RSet** to be shorthands for $\mathbf{Map}[\mathbf{HeapRef}, \mathbf{Any}]$ and $\mathbf{Set}[\mathbf{HeapRef}]$, respectively.

We define two translation relations that take types T, resp. well-typed terms t, and produce their lowered counterparts. The translation relation for types, $\mathsf{T} \triangleright \mathsf{T}'$, witnesses the erasure of type T to T'; for instance, if Cell is a heap type, then $\mathbf{Set}[\mathsf{Cell}] \triangleright \mathbf{Set}[\mathbf{HeapRef}]$. The translation relation for terms is notated as $\mathsf{h}, \rho, \mu; \Gamma \vdash \mathsf{t} \triangleright \mathsf{t}'$ and depends on a locally-mutable heap variable h, its reads and modifies domains, ρ and μ, and the typing environment Γ. When implicitly clear or the same in all occurrences, we omit h, ρ, μ and Γ and simply write $\mathsf{t} \triangleright \mathsf{t}'$. We assume the existence of a typing relation $\Gamma \vdash \mathsf{t} : \mathsf{T}$ and also omit Γ when it is clear from the context.

The encoding proceeds by translating each definition of an ADT D, heap type C, or function f in the surface program to a corresponding lowered definition. The data type definitions of the encoded program are obtained by taking all of the ADT definitions D with argument types erased by $\mathsf{T} \triangleright \mathsf{T}'$, and additionally introducing one single-constructor ADT for each heap type C (also with its field types erased). We refer to the resulting lowered ADTs as D_D and D_C. For each function definition $\mathbf{def}\ \mathsf{f}(\overline{\mathsf{x} : \mathsf{T}}) : \mathsf{S} = \{\mathbf{reads}(\mathsf{t}_\rho); \mathbf{modifies}(\mathsf{t}_\mu); \mathsf{t}\}$ in the original program we introduce two functions f and f_shim in the encoded program. The encoded function f takes the pre-state as an additional argument, and returns the resulting post-state along with its result value, yielding

$$\mathbf{def}\ \mathsf{f}(\mathsf{h}_0 : \mathbf{HeapMap}, \overline{\mathsf{x} : \mathsf{T}'}) : (\mathsf{S}', \mathbf{HeapMap}) = \{\mathbf{let}\ \rho = \mathsf{t}'_\rho\ \mathbf{in}\ \mathbf{let}\ \mu = \mathsf{t}'_\mu\ \mathbf{in}\ \mathsf{t}'\}$$

where $\mathsf{h}_0, \rho, \mu; \Gamma_0 \vdash \mathsf{t} \triangleright \mathsf{t}'$, as well as $\mathsf{h}_0, \rho, \emptyset; \Gamma_0 \vdash \mathsf{t}_s \triangleright \mathsf{t}'_s$ for $s \in \{\rho, \mu\}$, $\Gamma_0 = \overline{\mathsf{x} : \mathsf{T}}$, $\overline{\mathsf{T}} \triangleright \overline{\mathsf{T}'}$ and $\mathsf{S} \triangleright \mathsf{S}'$. Its companion, f_shim, encapsulates both the assumption of frame conditions and the checking of the associated heap contracts at each call site of f:

$$\begin{aligned}
&\mathbf{def}\ \mathsf{f}_\mathsf{shim}(\mathsf{h}_0 : \mathbf{HeapMap}, \rho_\mathsf{dom} : \mathbf{RSet}, \mu_\mathsf{dom} : \mathbf{RSet}, \overline{\mathsf{x} : \mathsf{T}'}) : (\mathsf{S}', \mathbf{HeapMap}) = \{ \\
&\quad \mathbf{let}\ \rho = \mathsf{t}'_\rho\ \mathbf{in}\ \mathbf{let}\ \mu = \mathsf{t}'_\mu\ \mathbf{in} \\
&\quad \mathbf{assert}(\rho \subseteq \rho_\mathsf{dom}); \mathbf{assert}(\mu \subseteq \mu_\mathsf{dom}); \\
&\quad \mathbf{let}\ \mathsf{y}_\mathsf{res} = \mathsf{f}(\mathsf{h}_0, \overline{\mathsf{x}})\ \mathbf{in} \\
&\quad \mathbf{let}\ \mathsf{y}_\mathsf{resR} = \mathsf{f}(\rho.\mathsf{mapMerge}(\mathsf{h}_0, \mathsf{dummyHeap}), \overline{\mathsf{x}})\ \mathbf{in} \\
&\quad \mathbf{assume}(\mathsf{y}_\mathsf{res}._1 = \mathsf{y}_\mathsf{resR}._1); \\
&\quad \mathbf{assume}(\mathsf{y}_\mathsf{res}._2 = \mu.\mathsf{mapMerge}(\mathsf{y}_\mathsf{resR}._2, \mathsf{y}_\mathsf{res}._2)); \\
&\quad \mathbf{assume}(\mathsf{y}_\mathsf{res}._2 = \mu.\mathsf{mapMerge}(\mathsf{y}_\mathsf{res}._2, \mathsf{h}_0)); \\
&\quad \mathsf{y}_\mathsf{res} \\
&\}
\end{aligned}$$

As an optimization, we omit the parts of the encoding that relate to the post-state when the **modifies** clause is empty. When the **reads** clause is empty as well, we avoid changing the function's signature altogether, so that pure functions remain pure.

The crucial rules of $\mathsf{t} \triangleright \mathsf{t}'$ are listed in Figure 7. Both FieldReadI and FieldUpdateI deal with field accesses of immutable data types and do not require interaction with the heap. In general, pure constructs are left untouched and their translation rules merely map over subexpressions. Imperative constructs, on the other hand, read or modify the locally-mutable heap h and refer to ρ and μ to enforce the heap contracts. For instance, FieldReadM handles field reads from a heap type C. It translates a read $\mathsf{t.f}$ to an assertion that the receiver object is in the reads set ($\mathsf{t}' \in \rho$), after which the object state is read from the heap ($\mathsf{h}\,[\mathsf{t}']$) and downcast to the corresponding lowered data type $\mathsf{D_C}$, from which the actual value is then projected ($\mathsf{x.f}$). The rule for function calls, Call, merely rewrites invocations of f to invocations of $\mathsf{f_{shim}}$, passing in the current heap h and the domains on which the callee is permitted to read and modify the heap. We always inline these shim functions, so the assertions in $\mathsf{f_{shim}}$ are effectively lifted to each call site of f and ensure that the reads and modifies clauses of the callee is subsumed by the caller's.

5.3 Quantifier-Free Frame Conditions

In the previous subsection we assumed a language construct called mapMerge that made it straightforward to express the necessary frame conditions. The crucial question that remains is how to lower mapMerge and its arguments to an efficiently decidable theory supported by an SMT solver. Our solution is to target the theory of (infinite, extensional) arrays in Z3, leveraging the fact that Stainless translates both sets and maps to such arrays. This means that reads and modifies expressions of type **Set**[**HeapRef**] become arrays typed HeapRef ⇒ Boolean, while heap maps of type **Map**[**HeapRef**, **Any**] are translated to HeapRef ⇒ Any. We can then use the array combinator $\mathrm{map}_f(a_1, \ldots, a_n)$ to express mapMerge efficiently. This array combinator is part of Z3's extended array theory [29] and axiomatized as $\forall i.\, \mathrm{map}_f(a_1, \ldots, a_n)[i] = f(a_1[i], \ldots, a_n[i])$. While the combinator can in practice only be applied to built-in functions, this is sufficient for our purposes: Given Stainless' encoding of sets and maps, one can use the if-then-else function ite of Z3, and translate $\mathsf{s.mapMerge}(\mathsf{m}_1, \mathsf{m}_2)$ as $\mathrm{map}_{\mathrm{ite}}([\mathsf{s}], [\mathsf{m}_1], [\mathsf{m}_2])$.

5.4 First-Class Heaps

A benefit of our encoding is that it naturally extends to explicit reasoning about alternative heap states within the program logic. Since our heaps are merely **Map**s, we can consider contexts with multiple heaps and express hyperproperties like determinism. Compare this to verifiers based on imperative languages, where relational verification requires constructions such as self-composition and product programs, limiting the applicability of existing toolchains [4, 14].

$$\frac{\mathsf{t} \triangleright \mathsf{t}' \quad \mathsf{t} : \mathsf{D}}{\mathsf{t.f} \triangleright \mathsf{t}'.\mathsf{f}} \qquad (\textsc{FieldReadI})$$

$$\frac{\mathsf{t}_1 \triangleright \mathsf{t}_1' \quad \mathsf{t}_2 \triangleright \mathsf{t}_2' \quad \mathsf{t}_1 : \mathsf{D}}{\mathsf{t}_1.\mathsf{f} := \mathsf{t}_2 \triangleright \mathsf{t}_1'.\mathsf{f} := \mathsf{t}_2'} \qquad (\textsc{FieldUpdateI})$$

$$\frac{\mathsf{t} \triangleright \mathsf{t}' \quad \mathsf{t} : \mathsf{C} \quad \mathsf{x} \text{ is fresh}}{\mathsf{t.f} \triangleright \mathbf{assert}(\mathsf{t}' \in \rho); \mathbf{let}\ \mathsf{x} = \mathsf{h}\,[\mathsf{t}']\ \mathbf{as}\ \mathsf{D_C}\ \mathbf{in}\ \mathsf{x.f}} \qquad (\textsc{FieldReadM})$$

$$\frac{\mathsf{t}_1 \triangleright \mathsf{t}_1' \quad \mathsf{t}_2 \triangleright \mathsf{t}_2' \quad \mathsf{t}_1 : \mathsf{C} \quad \mathsf{x} \text{ is fresh}}{\mathsf{t}_1.\mathsf{f} := \mathsf{t}_2 \triangleright \begin{array}{c}\mathbf{assert}(\mathsf{t}_1' \in \rho \cap \mu); \mathbf{let}\ \mathsf{x} = \mathsf{h}\,[\mathsf{t}_1']\ \mathbf{as}\ \mathsf{D_C}\ \mathbf{in} \\ \mathsf{h} := \mathsf{h.update}(\mathsf{t}_1', (\mathsf{x.f} := \mathsf{t}_2'))\end{array}} \qquad (\textsc{FieldUpdateM})$$

$$\frac{\overline{\mathsf{t}} \triangleright \overline{\mathsf{t}'} \quad \mathsf{x} \text{ is fresh}}{\mathsf{f}(\overline{\mathsf{t}}) \triangleright \mathbf{let}\ \mathsf{x} = \mathsf{f}_{\mathsf{shim}}(\mathsf{h}, \rho, \mu, \overline{\mathsf{t}'})\ \mathbf{in}\ \mathsf{h} := \mathsf{x}.\_2; \mathsf{x}.\_1} \qquad (\textsc{Call})$$

Fig. 7. Basic rules of the term translation relation $\mathsf{h}, \rho, \mu; \Gamma \vdash \mathsf{t} \triangleright \mathsf{t}'$. We abbreviate the relation as $\mathsf{t} \triangleright \mathsf{t}'$, since the omitted arguments are merely passed through by the above rules. The form $\mathbf{let}\ \mathsf{x} = \mathsf{t}_1\ \mathbf{as}\ \mathsf{T}\ \mathbf{in}\ \mathsf{t}_2$ is syntactic sugar for downcasts (see Section 5.2).

$$\begin{array}{lll} \text{Types} \ \ldots\ \mathsf{S}, \mathsf{T} & := & \ldots \mid \mathbf{Heap} \\ \text{Terms} \ \ldots\ \mathsf{t} & := & \ldots \mid \mathsf{Heap.get} \mid \mathsf{t.eval(t)} \mid \mathsf{Heap.unchanged(t, t, t)} \end{array}$$

$$\frac{}{\mathsf{Heap.get} \triangleright \rho.\mathsf{mapMerge}(\mathsf{h}, \mathsf{dummyHeap})} \qquad (\textsc{HeapGet})$$

$$\frac{\mathsf{t_h} \triangleright \mathsf{t_h'} \quad \mathsf{h}' \text{ is fresh} \quad \mathsf{h}', \mathsf{U}, \mathsf{U}; \Gamma \vdash \mathsf{t_e} \triangleright \mathsf{t_e'}}{\mathsf{t_h.eval(t_e)} \triangleright \mathbf{let\ var}\ \mathsf{h}' = \mathsf{t_h'}\ \mathbf{in}\ \mathsf{t_e'}} \qquad (\textsc{HeapEval})$$

$$\frac{\mathsf{t_s} \triangleright \mathsf{t_s'} \quad \mathsf{t_{h1}} \triangleright \mathsf{t_{h1}'} \quad \mathsf{t_{h2}} \triangleright \mathsf{t_{h2}'}}{\mathsf{Heap.unchanged}(\mathsf{t_s}, \mathsf{t_{h1}}, \mathsf{t_{h2}}) \triangleright \mathsf{t_{h1}'} = \mathsf{t_s'.mapMerge}(\mathsf{t_{h2}'}, \mathsf{t_{h1}'})} \qquad (\textsc{HeapUnchanged})$$

Fig. 8. Syntax of the surface language with first-class heaps and related term translation rules. The symbol U denotes the universal set of all **HeapRef**s.

The syntax extensions related to first-class heaps are shown in Figure 8 alongside the additional translation rules. The type translation simply erases **Heap** ▷ **HeapMap**. All of the new constructs are straightforward to encode in our scheme. Heap.get exposes the currently readable heap (HeapGet). We reduce $\mathsf{t_h.eval(t_e)}$ to translating $\mathsf{t_e}$ in the context of a fresh heap variable initialized

Benchmark	#LoC	#VCs	T	C	HC
Empty	10	0	6.3	0.0	0.0
AllocatorMono	73	80	12.8	4.1	0.8
ArraySimple	38	16	7.6	0.6	0.2
CellArraySimple	21	9	7.2	0.4	0.1
FibCache	38	32	11.1	2.8	0.3
MutList	81	148	46.2	35.4	2.2
MutListSetsOnly	45	54	30.3	22.4	1.4
NodeCycle	72	69	12.0	4.0	0.2
Queue	190	290	36.6	20.5	3.6
Stack	66	62	10.5	2.6	0.7
StackSimple (Fig. 1)	27	26	8.3	1.1	0.1
TaskParallel	46	38	8.3	1.1	0.2
TaskParallelBasic	58	51	8.6	1.2	0.2
TraitsReadsWrites	39	33	7.8	0.8	0.2
TreeImmutMapGeneric (Fig. 3)	55	33	17.1	8.3	0.2
UpCounter	48	32	8.0	0.9	0.2

Fig. 9. Evaluation results. For each benchmark we list the # of verification conditions discharged, the # lines of Scala code (including annotations), the total runtime T, the time spent checking VCs C, and the particular amount of time spent on VCs of heap contracts HC. Timings are given in seconds.

to $\mathsf{t_h}$ (HeapEval). Notably, during this translation we do not inject any further checks of **reads** and **modifies** by setting ρ and μ to the sentinel value U (denoting the universal set). While the lack of checks allows for reads outside a heap's original domain, they are well-defined (i.e., they equal the dummyHeap on those locations). Finally, $\mathsf{Heap.unchanged}(\mathsf{t_s}, \mathsf{t_{h1}}, \mathsf{t_{h2}})$ translates to an equality that holds iff for all objects in $\mathsf{t_s}$ the heaps $\mathsf{t_{h1}}$ and $\mathsf{t_{h2}}$ agree. The corresponding lowering rule HeapUnchanged leverages mapMerge in a way similar to our encoding of frame conditions. Namely, we take $\mathsf{t'_s.mapMerge}(\mathsf{t'_{h2}}, \mathsf{t'_{h1}})$ (the heap which interprets all objects as $\mathsf{t'_{h1}}$, except those in $\mathsf{t'_s}$, which it interprets as in $\mathsf{t'_{h2}}$), and require that it equals $\mathsf{t'_{h1}}$ itself.

6 Evaluation

We used our system to verify a number of benchmarks ranging in size and complexity. Among the examples we developed are both shallowly and deeply mutable data structures, a model of an object allocator, and a parallelization primitive for the fork-join model. In Figure 9 we summarize these benchmarks quantitatively in terms of total lines of code, and the time our system takes to verify the example. In particular, we report T, the total wall time elapsed when running an individual benchmark, which includes the time it takes the Scala compiler to process both our standard library and the benchmark, our extraction pipeline to lower from imperative Scala code to the functional fragment, and the time

spent on generating and checking verification conditions. The latter component is reported separately as *C*, and the time thereof spent on checking heap contracts as *HC*. The reported numbers were obtained on a machine with an AMD Ryzen 3700X 8-core CPU @ 3.6GHz and 32GB of RAM running Ubuntu 20.04, and using Z3 version 4.8.12. We explicitly list an *empty* benchmark that entails no verification conditions, but provides a baseline for the time spent on JVM startup, and, more importantly, extraction through the traditional Scala compilation pipeline plus various lowerings in Stainless before the actual generation and solving of VCs. We next discuss our experience using the tool and elaborate on some of the benchmarks listed.

Shallowly-Mutable Data Structures. We first consider "shallowly-mutable" data structures such as Cell[T] seen in Section 3 whose mutable data is stored directly in its fields, i.e., without any indirection. They provide a simple baseline for our system and play an important role as building blocks for larger data structures such as trees and arrays with fine-grained separation properties. However, shallowly-mutable data structures are useful in their own right: For instance, we implemented UpCounter which tracks a monotonically increasing variable and maintains an invariant relative to the counter's initial value. We also implemented a simple array (ArraySimple) and stack (StackSimple) which essentially act as wrappers around functional data structures in that they only store the reference to the head of an immutable list. For instance, ArraySimple[T] consists of a single mutable field **var** list: List[T]. In our examples we show safety wrt. bounds checks and non-emptyness when popping an element off the stack. We found that our system easily deals with this kind of mutability, requiring no additional proof hints whatsoever, in particular since the associated operations typically require no recursion through stateful functions, making them straightforward to verify and invalidate with counter-examples.

Mutable Linked Lists and Queues. As an example of a more complex data structure we implemented multiple variations of a mutable, acyclic, singly-linked list. We focussed on an append operation, which takes two valid linked lists l1 and l2 with disjoint representations and concatenates them, leaving l1 in a valid state. This is challenging in a system without a built-in notion of lists or trees, since establishing the well-formedness of lists (e.g., the absence of cycles) requires knowledge of heap separation and an inductive proof that maintains the property for intermediate nodes.

We considered several options to track a node's representation repr. One could express repr as a recursive function as in Section 3, or, instead, as a mutable **@ghost** field on each node. In our benchmarks we present two variants of the latter approach: MutList encodes the ghost field repr as List[AnyHeapRef], which has the added benefit of allowing predicates like valid to recurse on the representation, and can be converted to a Set[AnyHeapRef] as required by our **reads** and **modifies** clauses. MutListSetsOnly instead implements repr as Set[AnyHeapRef], whose encoded form requires no further conversion to interact with the mapMerge primitive we use for framing.

```
1  abstract class Task {
2    @ghost def readSet: Set[AnyHeapRef]
3    @ghost def writeSet: Set[AnyHeapRef] = { ??? } ensuring (_ ⊆ readSet)
4
5    def run(): Unit = { reads(readSet); modifies(writeSet); ??? : Unit }
6  }
7
8  def parallel(task1: Task, task2: Task): Unit = {
9    reads(task1.readSet ++ task2.readSet)
10   modifies(task1.writeSet ++ task2.writeSet)
11   require((task1.writeSet ∩ task2.readSet == ∅) &&
12           (task2.writeSet ∩ task1.readSet == ∅))
13   task1.run(); task2.run() // task1 and task2 complete before this function returns
14 }
```

Fig. 10. An interface for asynchronous computations and a sequential specification for fork-join parallelism. The ??? denotes unimplemented code in abstract classes.

We used a similar approach to implement Queue, which provides constant-time enqueue and dequeue methods using references to the first and last nodes. Given a valid queue we prove that enqueue and dequeue maintain validity and are functionally correct with respect to a serialized representation similar to toList in Section 3. The example demonstrates how safety properties can be established even in the presence of sharing and arbitrarily deep data structures.

The NodeCycle example illustrates how to define the inductive heap predicate for a cyclic list. We also establish that the prepend operation on such a list maintains cyclicity. Both this and the aforementioned example leverage first-class heaps to carry out the inductive proofs showing that the corresponding heap predicates continue to hold after modifications to the data structure.

Slices and Monolithic Arrays. Arrays are one of the most common data structures found in imperative code and thus a worthwhile target for verification. When specifying algorithms involving arrays it often pays to introduce slices, i.e., subarrays, as a means of abstraction. By extending the ArraySimple example we arrived at ArraySlice which provides safe indexing, update and re-slicing operations wrt. an underlying array. In the absence of sharing, this solution of encapsulating all array state in a single "monolithic" mutable heap object (the underlying array) is the natural and practical choice.

Fork-Join Parallelism. Since dynamic frames in our system are simply given by read-only expressions, users may define their own imperative abstractions. For instance, in TaskParallel we demonstrate how one can specify a primitive modelling fork-join parallelism. Figure 10 shows an excerpt introducing the Task interface that encapsulates an asynchronous computation and declares the set of heap objects that may be read and modified in the process. Further below we define the parallel(t1, t2) construct [23] itself, imposing a number of restrictions:

Firstly, callers of parallel have to establish accessibility to both t1 and t2's frames (lines 9-10). Secondly, we require that the read set of t1 is disjoint from t2's write set and vice-versa (lines 11-12). This separation property justifies replacing our sequential model of parallel by a more efficient runtime implementation executing the two tasks concurrently. Users can define new asynchronous tasks by implementing Task. Operations such as those on cell-based data structures discussed above are straightforward to parallelize in this way. Our introductory example of Section 3 could be parallelized by defining a new class TMapTask[T](t: Tree[T], f: T ⇒ T) whose run method calls tmap, and replacing the recursive calls in tmap by parallel(TMapTask(left, f), TMapTask(right, f)).

7 Conclusions

We have presented an approach that extends the Stainless verifier with support for shared mutable data. Our goal was to preserve as much as possible certain features of Stainless that we consider useful: the ability to reason about purely functional programs efficiently and the ability to report counterexamples. Our strategy to report counterexamples is to avoid the use of quantifiers. This is by no means the only possibility, as witnessed by the success of approaches that use them effectively. Yet we believe that the use of decision procedures in the long term results in a more predictable verification experience than direct use of general quantifiers. Our experiments suggest that the approach holds promise, even though the performance of map operators indicates that they nonetheless require non-trivial reasoning in the Z3 solver.

An integration of insights from verifiers and proof frameworks based on separation logic is a promising direction to potentially improve usability of our approach. SMT-LIB notations and competitions for separation logic [36] are likely to be a useful resource for this task, even if these benchmarks typically do not focus on reasoning about as detailed functional correctness properties as our examples. Another direction for improving automation is inductive reasoning, both for separation logic predicates themselves [39] and for pure recursive functions [35].

In conclusion, our paper makes the initial case for an approach that is semantically simple and promises to be predictable. We hope that it will motivate both the SMT solver builders and verification tool builders to work jointly to improve the performance, the predictability, and the ability to report counterexamples for verification, with array theories being among the most promising future directions [8, 11, 29, 38].

Acknowledgments

We thank Antoine Brunner for helping implement a first prototype in Stainless, and the anonymous reviewers for their valuable feedback. This work is supported by the Swiss National Science Foundation project number 200021_175676.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
2. Ahman, D., Hriţcu, C., Maillard, K., Martínez, G., Plotkin, G., Protzenko, J., Rastogi, A., Swamy, N.: Dijkstra monads for free. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 515–529 (2017)
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: International Symposium on Formal Methods. pp. 200–214. Springer (2011)
5. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Asian Symposium on Programming Languages and Systems. pp. 52–68. Springer (2005)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
7. Blanc, R.W.: Verification by Reduction to Functional Programs. Ph.D. thesis, EPFL, Lausanne (7 2017). https://doi.org/10.5075/epfl-thesis-7636, http://infoscience.epfl.ch/record/230242
8. Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 427–442. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11609773_28
9. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
10. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
11. Daca, P., Henzinger, T.A., Kupriyanov, A.: Array folds logic. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 230–248. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_13
12. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019). https://doi.org/10.1145/3338112
13. Distefano, D., Parkinson J, M.J.: jstar: Towards practical verification for java. ACM Sigplan Notices **43**(10), 213–226 (2008)
14. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **42**(1), 1–37 (2019)
15. Filliâtre, J.C.: Verification of non-functional programs using interpretations in type theory. Journal of Functional Programming **13**(4), 709–745 (2003). https://doi.org/10.1017/S095679680200446X
16. Finkbeiner, B.: Model checking algorithms for hyperproperties (invited paper). In: Henglein, F., Shoham, S., Vizel, Y. (eds.) Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen,

Denmark, January 17-19, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12597, pp. 3–16. Springer (2021). https://doi.org/10.1007/978-3-030-67067-2_1
17. Hamza, J., Voirol, N., Kunčak, V.: System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang **OOPSLA** (November 2019). https://doi.org/10.1145/3360592
18. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Miller, E.L., Hand, S. (eds.) Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015. pp. 1–17. ACM (2015). https://doi.org/10.1145/2815400.2815428
19. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for c and java. In: NASA Formal Methods Symposium. pp. 41–55. Springer (2011)
20. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, e20 (2018). https://doi.org/10.1017/S0956796818000151
21. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4085, pp. 268–283. Springer (2006). https://doi.org/10.1007/11813040_19
22. Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. p. 211–222. CCS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2508859.2516721
23. Kuncak, V., Prokopec, A.: Parallel programming (Lecture 1.4: Running computations in parallel). EPFL Courseware (February 2018), https://courseware.epfl.ch/courses/course-v1:EPFL+parprog1+2018_T1/about and https://www.youtube.com/watch?v=DbVt8C0-Oe0
24. Leino, K.R.M.: This is Boogie 2. Manuscript KRML **178**(131), 9 (2008)
25. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
26. Moggi, E.: Notions of computation and monads. Information and computation **93**(1), 55–92 (1991)
27. de Moura, L.M., Bjørner, N.: Model-based theory combination. Electron. Notes Theor. Comput. Sci. **198**(2), 37–49 (2008). https://doi.org/10.1016/j.entcs.2008.04.079
28. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24

29. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA. pp. 45–52. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351142
30. Müller, P., Schwerhoff, M., Summers, A.J.: Automatic verification of iterated separating conjunctions using symbolic execution. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 405–425. Springer International Publishing (2016)
31. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
32. Odersky, M., Spoon, L., Venners, B.: Programming in Scala, Fourth Edition (A comprehensive step-by-step guide). Artima (2019), https://www.artima.com/shop/programming_in_scala_4ed
33. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2142, pp. 1–19. Springer (2001). https://doi.org/10.1007/3-540-44802-0_1
34. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. Log. Methods Comput. Sci. **8**(3) (2012). https://doi.org/10.2168/LMCS-8(3:1)2012
35. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2015)
36. Sighireanu, M., Pérez, J.A.N., Rybalchenko, A., Gorogiannis, N., Iosif, R., Reynolds, A., Serban, C., Katelaan, J., Matheja, C., Noll, T., Zuleger, F., Chin, W., Le, Q.L., Ta, Q., Le, T., Nguyen, T., Khoo, S., Cyprian, M., Rogalewicz, A., Vojnar, T., Enea, C., Lengál, O., Gao, C., Wu, Z.: SL-COMP: competition of solvers for separation logic. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III. Lecture Notes in Computer Science, vol. 11429, pp. 116–132. Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_8
37. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. **34**(1), 2:1–2:58 (2012). https://doi.org/10.1145/2160910.2160911
38. Stump, A., Barrett, C., Dill, D., Levitt, J.: A decision procedure for an extensional theory of arrays. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. pp. 29–37. IEEE Comput. Soc, Boston, MA, USA (2001). https://doi.org/10.1109/LICS.2001.932480
39. Ta, Q., Le, T.C., Khoo, S., Chin, W.: Automated mutual induction proof in separation logic. Formal Aspects Comput. **31**(2), 207–230 (2019). https://doi.org/10.1007/s00165-018-0471-5
40. Voirol, N.C.Y.: Verified Functional Programming. Ph.D. thesis, EPFL, Lausanne (2019). https://doi.org/10.5075/epfl-thesis-9479, http://infoscience.epfl.ch/record/268824
41. Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming. pp. 61–78 (1990)