

Mechanized Verification with Sharing

Gregory Malecha

Greg Morrisett

Harvard University SEAS

Abstract. We consider software verification of imperative programs by theorem proving in higher-order separation logic. Of particular interest are the difficulties of encoding and reasoning about sharing and aliasing in pointer-based data structures. Both of these are difficulties for reasoning in separation logic because they rely, fundamentally, on non-separate heaps. We show how sharing can be achieved while preserving abstraction using mechanized reasoning about fractional permissions in Hoare type theory.

1 Motivation

Axiomatic semantics [7] is one way to formally reason about programs. Under these semantics, programs are analyzed by considering the effect of primitive operations on predicates over the heap. Unfortunately, stating and reasoning about these predicates is complicated due to potential pointer aliasing. It was not until Reynolds proposed separation logic [16] that reasoning about imperative programs in a modular way became tractable. However, even with this logic some specifications are still not simple. For example, many algorithms are simplified by sharing data which can be difficult to express in separation logic.

The difficulty comes from conflicting goals: We want to reason locally and compositionally about programs, and, at the same time, we wish to share data globally to make algorithm and data structure implementations more efficient. Vanilla separation logic provides the first, but makes the second difficult because of the non-local effects illustrated by the following Java program:

```
1 void error(List<T> lst) {  
    Iterator<T> itr = lst.iterator();  
3  lst.remove(0);  
    itr.next(); // throws ConcurrentModificationException  
5}
```

Here, line 3 has removed the element that the iterator is referencing, so we've destroyed the view that the iterator is abstracting even though line 3 does not even mention the iterator. If problems like this go undetected at run-time, they can result in `NullPointerExceptions` in Java, or memory corruption or segmentation faults in lower level languages such as C.

In this paper we show how type-directed formal verification can be used to verify data structures that share state, in particular collections and their iterators. Our data structures are heap-allocated and make liberal use of pointer

aliasing. We have found that sharing makes formally reasoning about the correctness of programs in an automated way difficult, and we believe general theorem proving techniques are most suitable to address these problems that other techniques have not been able to.

We consider sharing of two sorts, *external* and *internal*. In *external* sharing, we wish to support multiple, simultaneous views of the same underlying memory for clients. In *internal* sharing, the sharing is completely hidden behind the abstraction allowing the client to reason using a simple interface while the implementation uses aliasing to make implementations more efficient.

Contributions

We begin with a brief overview of the Ynot verification library [4] (Section 2), demonstrating how higher-order separation logic can be used to provide abstraction. We then cover our contributions, we

- Show how fractional permissions [8] can be applied to provide sharing of high-level abstractions, we focus on collections. (Section 3)
- Show how external sharing can be leveraged to mechanically verify higher-order, effectful computations in Ynot, we focus on iterators. (Section 4)
- Show how internal sharing can be expressed by describing the representation of B+ tree, skirted n -ary trees, and how our approach simplifies the implementation of an iterator. (Section 5)
- While formalizing B+ trees, we also show a technique for formalizing data structures with a non-functional connection to their specification. (Section 5)

In our presentation, we focus on interfaces in stylized Coq, but our implementation and verification are available at <http://ynot.cs.harvard.edu/>. After our contributions, we consider the burden of verification, the implications of our techniques, and related work (Section 6).

We believe that our methodology extends previous work describing aliasing in separation logic [3] by being amenable to machine-checkable proofs and embeddable in Hoare-type theory. Previous work has developed paper-and-pencil proofs and, as has been seen in other contexts [1], the evolution from rigorous, manual proofs to mechanically verified proofs is not always straightforward.

2 Background

Ynot [4] is a Coq library that implements Hoare type theory [14] to reason about imperative programs using types. Hoare logic describes commands using Hoare triples, commands along with pre- and post-conditions. Ynot encodes these in the type of the `Cmd` monad.

$$\{P\}c\{r \Rightarrow Q\} \quad \equiv \quad c : \text{Cmd } (P) (r \Rightarrow Q)$$

where the command c has pre-condition P and post-condition Q that depends on the return value of c (bound to r). This type means that the command c can

be run in any state that satisfies P and, if c terminates with value r , then the resulting state will satisfy Qr .

Ynot defines pre- and post-conditions in the logic of Coq as predicates over heaps, which, themselves, are defined as functions from pointers to optional values. Previous work [4] showed how using a stylized fragment of separation logic makes verification conditions more amenable to automation and therefore less burdensome for the programmer to prove. As in previous work, we use a shallow embedding of separation logic which we extend with support for fractional permissions (Figure 1).

$h \models P$ Heap Propositions (*hprop*)

Empty	$h \models \mathbf{emp}$	$\iff \Delta$	$\forall p. h\ p = \mathbf{None}$
Points-to	$h \models p \overset{q}{\mapsto} v$	$\iff \Delta$	$h\ p = \mathbf{Some}(q, v) \wedge \forall p'. p \neq p' \rightarrow h\ p = \mathbf{None}$
Separating Conjunction	$h \models P * Q$	$\iff \Delta$	$\exists h_1\ h_2. P\ h_1 \wedge Q\ h_2 \wedge h = h_1 \uplus h_2 \wedge h_1 \perp h_2$
Existentials	$h \models \exists x. P_x$	$\iff \Delta$	$\exists x. P_x\ h$
Pure Injection	$h \models [p]$	$\iff \Delta$	$\mathbf{emp}\ h \wedge p$

$h_0 \perp h_1$ Heap Disjointness

$$h_0 \perp h_1 = \forall p. \begin{cases} v_0 = v_1 \wedge q_0 + q_1 \leq 1 & \forall i \in \{0, 1\}. h_i\ p = \mathbf{Some}(q_i, v_i) \\ q_i \leq 1 & i \in \{0, 1\} \wedge h_i\ p = \mathbf{Some}(q_i, v) \wedge h_{1-i}\ p = \mathbf{None} \\ \mathbf{True} & \forall i \in \{0, 1\}. h_i\ p = \mathbf{None} \end{cases}$$

$h_0 \uplus h_1$ Heap Union

$$(h_0 \uplus h_1)\ p = \begin{cases} \mathbf{Some}(q_0 + q_1, v) \leq 1 & \forall i \in \{0, 1\}. h_i\ p = \mathbf{Some}(q_i, v) \\ \mathbf{Some}(q_i, v) & i \in \{0, 1\} \wedge h_i\ p = \mathbf{Some}(q_i, v) \wedge h_{1-i}\ p = \mathbf{None} \\ \mathbf{None} & \forall i \in \{0, 1\}. h_i\ p = \mathbf{None} \end{cases}$$

Fig. 1. The shallow embedding of separation logic used in Ynot.

The empty heap (\mathbf{emp}) denotes a heap containing no allocated cells, all pointers are mapped to \mathbf{None} ¹. The permission to access the heap cell pointed to by p is given by the fractional points-to relation, $p \overset{q}{\mapsto} v$ [8,6,15]. We use the simple model of fractional permissions originally developed by Boyland [8]. In this work, the value of q is a rational number such that $0 < q \leq 1$, in all cases the points-to relation asserts that the heap contains a cell with the value v pointed to by p . When $q = 1$, the points to assertion gives code the ability to read, write, and deallocate the cell. When $q < 1$, the points-to relation gives read-only access to

¹ The symbols \mathbf{None} and \mathbf{Some} are the constructors of the `option` α type which represents an optional value of type α which is included in the `Some` constructor.

the heap cell. The separating conjunction ($*$) states that the two conjuncts hold on two “disjoint” pieces of the heap. In the definition $h_0 \perp h_1$ defines the disjointness which is slightly complicated by the fractional permissions. Two heaps are disjoint if each pointer is mapped by only one heap or the values are the same and the fractions sum to a valid fraction. The \uplus operator defines a similar notion of unioning disjoint heaps. Ynot also supports existential quantification and pure propositions (propositions that do not mention the heap such as $x = y$ or $x < 5$) in heap propositions.

```

new   :  $\Pi(T : \mathbf{Type})(v : T), \mathbf{Cmd}(\mathit{emp})(p : \mathit{ptr} \Rightarrow p \mapsto v)$ 
free  :  $\Pi(p : \mathit{ptr}), \mathbf{Cmd}(\exists T, \exists v : T, p \mapsto v)(\_ : \mathit{unit} \Rightarrow \mathit{emp})$ 
read  :  $\Pi(T : \mathbf{Type})(p : \mathit{ptr})(P : T \rightarrow \mathit{hprop}),$ 
          $\mathbf{Cmd}(\exists v : T, p \stackrel{q}{\mapsto} v * P v)(v : T \Rightarrow p \stackrel{q}{\mapsto} v * P v)$ 
write :  $\Pi(T : \mathbf{Type})(p : \mathit{ptr})(v : T), \mathbf{Cmd}(\exists T, \exists v' : T, p \mapsto v')(\_ : \mathit{unit} \Rightarrow p \mapsto v)$ 
bind  :  $\Pi(T U : \mathbf{Type})(P P' : \mathit{hprop})(Q : T \rightarrow \mathit{hprop})(Q' : U \rightarrow \mathit{hprop}),$ 
          $(\forall v : T, Q v \Longrightarrow P') \rightarrow \mathbf{Cmd}(P)(Q) \rightarrow (T \rightarrow \mathbf{Cmd}(P')(Q')) \rightarrow \mathbf{Cmd}(P)(Q')$ 
return:  $\Pi(T : \mathbf{Type})(v : T), \mathbf{Cmd}(\mathit{emp})(r : T \Rightarrow [r = v])$ 
cast  :  $\Pi(T : \mathbf{Type})(P P' : \mathit{hprop})(Q Q' : T \rightarrow \mathit{hprop}), (P' \Longrightarrow P) \rightarrow$ 
          $(\forall v : T, Q v \Longrightarrow Q' v) \rightarrow \mathbf{Cmd}(P)(v : T \Rightarrow Q v) \rightarrow \mathbf{Cmd}(P')(v : T, Q' v)$ 
frame :  $\Pi(T : \mathbf{Type})(P Q R : \mathit{hprop}), \mathbf{Cmd}(P)(r : T)(Q r) \rightarrow$ 
          $\mathbf{Cmd}(P * R)(r : T \Rightarrow Q r * R)$ 

```

Fig. 2. Axiomatic basis for Hoare type theory using separation logic.

Ynot axiomatizes the primitive heap operations using the commands given in Figure 2. The **new** command allocates memory by producing the read-write capability to access the memory cell pointed to by the return value. The pre-condition specifies that the command needs no heap capabilities so the resulting pointer must be globally unique. The **free** command deallocates a memory cell by consuming the read-write permission to access the cell. The **read** command reads the values from a cell given a predicate, P , that describes the rest of the heap based on this value. The dependence on P allows us to enforce that the v in the pre-condition is the same as the v in the post-condition because P could include a precise equation on v . For example, if p pointed to a pointer to v , we could pick $P = \mathbf{fun} r \Rightarrow r \mapsto v$ thus making the post-condition reduce to $p \stackrel{q}{\mapsto} r * r \mapsto v$. The **write** command updates the value in a heap cell given a pointer and the new value.

These commands are combined using monadic **bind** and **return** in addition to a **cast** command that takes a proof and applies Hoare’s consequence rule. The **frame** command extends the footprint of a command with extra capabilities that are invariant under the command. This is essential to local reasoning and enables Ynot to run a command with pre-condition P and post-condition Q in an environment satisfying $P * R$ and allows us to infer the post-condition $Q * R$.

3 Sharable Abstractions: Linked Lists

In this section, we develop the basis of our contributions by defining a simple interface for externally sharable list structures. Sharing will allow multiple read-only views of the list or a single read-write view. We will achieve this using fractional permissions in the same way that we do for heap cells.

In Ynot, abstract data types are defined by a representation predicate and associated theorems and imperative commands. The interface for sharable lists (`Implist`) is given as a type-class [18] in Figure 3. The class is parametrized

```

2  Fixpoint specNth {T} (ls : list T) (n : nat) : option T :=
3  match ls, n with
4    | nil, _      => None
5    | a :: _, 0   => Some a
6    | _ :: b, S n => specNth b n
7  end

8  Class Implist (T : Type) (tlst : Type) := {
9    (* The tlst is a handle to perm capabilities to the list T *)
10   llist : perm → tlst → list T → hprop ;
11   (* Fractional merging and splitting of lists *)
12   llist_split : ∀ q q' t m, q |#| q' →
13     llist (q + q') t m ⇔ llist q t m * llist q' t m ;
14   (* Allocate an empty list *)
15   new      : Cmd (emp) (res : tlst ⇒ llist 1 res nil) ;
16   (* Free the list *)
17   free    : II (t : tlst),
18     Cmd (∃ ls : list T, llist 1 t ls) (_ : unit ⇒ emp) ;
19   (* Get the ith element from the list if it exists. *)
20   sub     : II (t : tlst T) (i : nat) (m : #list T#) (q : #perm#),
21     Cmd (llist q t m)
22     (res : option T ⇒ llist q t m * [res = specNth m i]) ;
23   (* Insert an element at the ith position in the list. *)
24   insert  : II (t : tlst) (v : T) (i : nat) (m : #list T#),
25     Cmd (llist 1 t m)
26     (_ : unit ⇒ llist 1 t (specInsert v i m)) ;
27 }

```

Fig. 3. Externally-sharable list interface.

by the type of the elements in the list (T) and the type of handles to the list (`tlst`). The representation predicate (`llist`) relates a fractional permission (of type `perm`), the list handle and a functional model of the list (the `list T`) to the imperative representation, i.e. the structure of the heap. The heap proposition `llist q t l` states that t is a handle to a q -fraction of an imperative representation of the functional list l . Conceptually, we can think of this as $t \xrightarrow{q} l$.

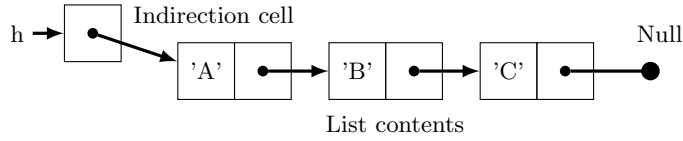


Fig. 4. A heap representing the list ['A', 'B', 'C'].

Assuming this, `new` and `free` are analogous to Ynot’s `new` and `free` commands. The specifications for `sub` and `insert` are expressed by relating their return value and post-condition to the result of pure functions (`specNth` and `specInsert`) that we take as specifications (we give the `specNth` function as an example of our specifications). We use the `#` in types to denote computationally irrelevant variables [4]. These can be thought of as compile-time-only values that are used to specify the behavior of computations without incurring run-time overhead.

One easy way to realize this interface is using singly-linked lists as shown in Figure 4. The following recursive equations specify the representation invariant for singly-linked list segments between pointers *from* and *to*.

$$\text{llseg } q \text{ from to nil} \stackrel{\Delta}{\iff} [from = to] \quad (1)$$

$$\text{llseg } q \text{ (Ptr } from) \text{ to } (a :: b) \stackrel{\Delta}{\iff} \exists x. from \stackrel{q}{\mapsto} \text{mkNode } a \ x * \text{llseg } q \ x \text{ to } b \quad (2)$$

$$\text{tlst } T = \text{ptr} \quad (3)$$

$$\text{lilst } q \ t \ ls \stackrel{\Delta}{\iff} \exists hd. t \stackrel{q}{\mapsto} hd * \text{llseg } hd \ \text{Null } ls \quad (4)$$

In equation (1), the model list is empty so the start and end pointers are the same. When the model list is not empty, i.e. it is a `cons (a :: b)`, *from* must not be null, and there must exist a pointer *x* such that *from* points to a heap cell containing *a* and *x* ($from \stackrel{q}{\mapsto} \text{mkNode } a \ x$) and *x* points to the rest of the list ($\text{llseg } x \text{ to } b$). Equation (4) makes the list mutable by making `tlst` an indirection pointer so the pointer to the head of the list can change.

Since the definition only claims a *q*-fraction of the list, all of the points-to assertions have fraction *q*. This allows us to prove the `lilst_split` lemma that states a $q_0 + q_1$ fraction of the list is equivalent to a q_0 fraction of the list disjoint from a q_1 fraction of the list. We can use this proof to create two disjoint, read-only views of the same list to share.

4 External Sharing: Iterators

The ability to share the list abstraction pays off when we need to develop another view of the list. Here, we develop a simple, efficient iterator over our list representation.

Our iterator is defined by a representation predicate (`lister`) and commands for creating iterators (`open`), advancing the iterator (`next`), and deallocating iterators (`close`):

```

Parameter titr : Type → Type.
2Parameter liter : ∀ T. perm → tlst T →
  titr → list T → nat → hprop.
4Parameter open : II (T : Type) (t : tlst T) (m : #list T#)
  (q : #perm#),
6  Cmd (l1ist q t m) (res : titr T ⇒ liter q t res m 0).
  Parameter next : II (T : Type) (t : titr T) (m : #list T#)
8  (idx : #nat#) (own : #tlst T#) (q : #perm#),
  Cmd (liter t m idx)
10  (res : option T ⇒ liter t m (idx + 1) *
    [res = specNth m idx]).
12Parameter close : II (T : Type) (t : titr T) (own : #tlst T#)
  (m : #list T#) (q : #perm#),
14  Cmd (∃ idx : tlst T, liter q own t m idx)
    (_ : unit ⇒ l1ist q own m).

```

The representation predicate defines the heap by relating a fractional ownership of the list and the handle to the underlying list to the iterator handle, the functional contents of the list, and a natural number which defines the current position in the list. Here, the fractional permission is the fractional ownership of the underlying list, not of the iterator, so even if this fraction is not 1, we will still be able to call `next`. The `open` computation constructs an iterator to the beginning of a `tlst T` by converting the heap predicate from `l1ist q t m` to `liter q t res m 0`. The `next` command returns the current element in the list (or `None` if the iterator is past the end of the list) and advances the position, reflected in the index argument of `liter`. The `close` command reverses the effect of `open` by converting the `liter` back into a `l1ist`.

The owner parameter to the representation predicate is necessary for describing the heap precisely enough to support the `close` command. Its use is similar the use of ownership types[5]. By making it a parameter we can specify that the `l1ist` permissions in the post-condition of `close` are exactly those that went into the `open` command.

With this, we can describe an iterator by full ownership of a heap cell containing a pointer to the current node, and fractional ownership of the owner pointer and underlying list. For simplicity implementing the interface, we break the specification of the list into two parts: the part that has already been visited (`firstn i m`) that goes from `st` to `cur`, and the rest (`skipn i m`) that goes from `cur` to `Null`.

$$\begin{aligned}
\text{titr } T &= \text{ptr} \\
\text{iter } \text{own } q \text{ t } m \text{ } i &\stackrel{\Delta}{\iff} \exists \text{cur}. \exists \text{st}. \text{own } \overset{q}{\mapsto} \text{st} * t \mapsto \text{cur} * \\
&\quad \text{llseg } q \text{ st } \text{cur} (\text{firstn } i \text{ } m) * \text{llseg } q \text{ cur } \text{Null} (\text{skipn } i \text{ } m)
\end{aligned}$$

5 Internal Sharing & Non-functional Heaps: B+ trees

We now turn to the problem of internal sharing. Recall that in internal sharing, we completely hide the sharing from the client. To demonstrate our tech-

nique, we discuss the representation of B+ trees that we presented in previous work [12]. We choose B+ trees to implement this interface because they have a structure that is tricky to reason about because of aliasing and previous work only demonstrated an imperative fold rather than the more primitive iterator. Our implementation for this interface does not include fractional permissions, though we believe that it would be relatively straightforward to add them.

```

2 (* tfmap is the type of finite maps from key to value. *)
2 Class FiniteMap (K V : Type) (tfmap : Type) := {
  (* The tfmap handle represents the fmap. *)
4  repMap : tfmap → fmap K V → hprop ;
  (* Create an empty finite map. *)
6  new : Cmd emp (h : tfmap ⇒ repMap h nil) ;
  (* Associate the key k with the value m. *)
8  insert : II (h : tfmap) (k : K) (v : V) (m : #fmap K V#),
  Cmd (repMap t m)
10      (res : option V ⇒ repMap h (specInsert v m) *
  [res = specLookup k m]) ;
12  (** ... free & lookup ... **)

14  titr : Type ;
  repIter : tfmap → titr → fmap K V → nat → hprop ;
16  open : II (h : tfmap) (m : #fmap K V#),
  Cmd (repMap h m) (res : titr ⇒ repIter h res m 0) ;
18  next : II (h : titr) (own : #tfmap#) (m : #fmap K V#)
  (idx : #nat#),
20  Cmd (repIter own h m idx)
  (res : option (K * V) ⇒
22      repIter own h m (idx + 1) * [specNth m idx]) ;
  close : II (h : titr) (own : #tfmap#) (m : #fmap K V#),
24  Cmd (∃i. repIter own h m i) (_ : unit ⇒ repMap own m)
}

```

Fig. 5. The imperative finite map interface.

Figure 5 gives our target interface for finite maps and their iterators, which we combine for brevity. The class is parametrized by the type of keys, values, and finite map handles. The logical model is a sorted association list (`fmap K V`) that we relate to the handle with the heap proposition `repMap q t m`. The remaining computations are similar to those of the list; we support allocation and deallocation as well as key lookup and key-value insertion. The iterator predicate is the same as the list iterator predicate except it does not have the fractional permission. The `open`, `next` and `close` commands are the same as for the list.

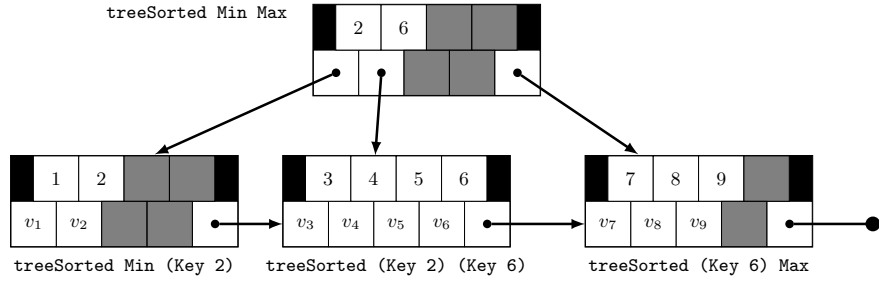


Fig. 6. An B+ tree of arity 4 ($n = 4$) for the finite map from $i \mapsto v_i$ for $1 \leq i \leq 9$.

B+ trees are balanced, ordered, n -ary trees that store data only at the leaves and maintain a pointer list in the fringe to make in-order iteration of the values efficient. Figure 6 shows a simple B+ tree with arity 4.

As with most tree structures, B+ trees are comprised of two types of nodes:

- Leaf nodes store data as a sequence of at most n key-value pairs in increasing order by key. The trailing pointer position points to the next leaf node.
- Branch nodes contain a sequence of at most n keys-subtree pairs and a final subtree. The pairs are ordered such that the keys in a subtree are less than or equal to the associated key (represented in the figure as **treeSorted min max**). For example, the second subtree can only contain values greater than 2 and less than or equal to 6. The final subtree covers the span greater than the last key; in the figure, this is the span greater than 6.

As with the iterator, the two main difficulties in formalizing B+ trees reveal themselves in the representation predicate. The first concerns the fact that multiple trees can represent the same finite map. The second concerns the aliasing at the leaves which is necessary to make iteration efficient.

The standard way to address the first problem is to use a direct relational specification of the heap, existentially quantifying the splitting of the list into subtrees at each level [17]. While this works well for paper-and-pencil proofs, it makes automation difficult because tactics need to guess the way that the heap is broken up at every step in order to instantiate existentials. Following this approach can yield goals with many existential variables that are not trivial to pick automatically. To avoid this, we factor the relation between the interface model and the heap description into a relation and a function, as shown in Figure 7.

Our representation model is a functional tree that we index by the height to enforce the balancedness constraint. In Coq, we could define this as follows, though we will modify it slightly to address the next problem:

```

Fixpoint ptree (h : nat) : Type :=
2 match h with
  | 0    => list (key * value)
4  | S h' => list (key * ptree h') * ptree h'

```

end

The second difficulty deals more directly with sharing. In the standard representation for a tree, we existentially quantify the pointers at the parent pointer for each node, but, if we follow this approach we can not directly encode the aliasing at the leaves because the predicate does not have access to both pointers. We could quantify the leaf pointers when the tree splits, but this gets ugly because we are working with n -ary trees. This would also lead to difficulties when defining iterators because we will want to frame the trunk part of the computation and consider only the leaves. Instead, we embed the pointers directly in the representation model using the following type:

```

1 Fixpoint ptree (h : nat) : Type :=
  ptr * match h with
3   | 0    => list (key * value)
   | S h' => list (key * ptree h') * ptree h'
5   end

```

Using this representation model, we can easily compute the pointers that alias without needing to worry about scoping since all of the pointers will be quantified at the root.

With this model, we can turn to describing the heap. We define `repTree h o p` to hold on to a heap representing the `ptree p` of height h when the rightmost leaf's next pointer equals o :

$$\begin{aligned}
 \text{repTree } 0 \text{ } \text{optr } (p', ls) &\stackrel{\Delta}{\iff} \exists \text{ary. } p' \mapsto \text{mkNode } 0 \text{ } \text{ary } \text{optr} * \text{repLeaf } \text{ary } ls \\
 \text{repTree } (1 + h) \text{ } \text{optr } (p', (ls, \text{next})) &\stackrel{\Delta}{\iff} \\
 \exists \text{ary. } p' \mapsto \text{mkNode } (h + 1) \text{ } \text{ary } (\text{ptrFor } \text{next}) * & \\
 \text{repBranch } \text{ary } (\text{firstPtr } \text{next}) \text{ } ls * \text{repTree } h \text{ } \text{optr } \text{next} &
 \end{aligned}$$

The `repTree` predicate has two cases depending on the `ptree`'s height. In the leaf case, the array holds the list of key-value pairs from the `ptree`.

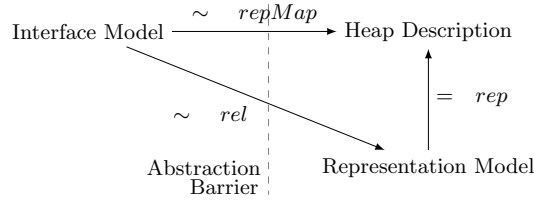
$$\begin{aligned}
 \text{repLeaf } \text{ary } [v_1, \dots, v_m] &\stackrel{\Delta}{\iff} \\
 \text{ary}[0] \mapsto \text{Some } v_1 * \dots * \text{ary}[m - 1] \mapsto \text{Some } v_m * & \\
 \text{ary}[m] \mapsto \text{None} * \dots * \text{ary}[n - 1] \mapsto \text{None} &
 \end{aligned}$$


Fig. 7. Decouple the relational mapping between the interface and the heap by factoring out a representation model that is functionally related to the heap.

In the branch case, the array holds key-pointer pairs such that each pointer points to the representation of the corresponding subtree in the `ptree`. This is captured by the `repBranch` predicate:

$$\begin{aligned} \text{repBranch } \text{ary } \text{optr } [(k_1, t_1), \dots, (k_m, t_m)] &\stackrel{\Delta}{\iff} \\ \text{ary}[0] &\mapsto \text{Some } (k_1, \text{ptrFor } t_1) * \text{repTree } h \text{ (firstPtr } t_2) t_1 * \dots * \\ \text{ary}[m-2] &\mapsto \text{Some } (k_{m-1}, \text{ptrFor } t_{m-1}) * \text{repTree } h \text{ (firstPtr } t_m) t_{m-1} * \\ \text{ary}[m-1] &\mapsto \text{Some } (k_m, \text{ptrFor } t_m) * \text{repTree } h \text{ optr } t_m * \\ \text{ary}[m] &\mapsto \text{None} * \dots * \text{ary}[n-1] \mapsto \text{None} \end{aligned}$$

At this point, we have defined the `rep` function from Figure 7; it remains to define `rel`. A standard relation would be fine to implement this, but since each tree corresponds to exactly 1 finite map, we can simplify things by computing the finite map (using `as_map`) associated with the tree and stating that it equals the desired model.

We can pick the handle type to be a pointer and define the full representation predicate to be the conjunction of `rep` and `rel` with some additional pure facts:

$$\begin{aligned} \text{repMap } \text{hdl } m &\stackrel{\Delta}{\iff} \exists h. \exists p : \text{ptree } h. \\ \text{hdl} &\mapsto (\text{ptrFor } p, \#p\#) * \text{repTree } h \text{ None } p * \\ &[m = \text{as_map } p] * [\text{treeSorted } h p \text{ MinMax}] \end{aligned}$$

By packing a copy of the `ptree` with the root pointer, we avoid the need to search for a model during proofs. The alternative is to show that there is at most one `ptree` that a given pointer and heap can satisfy (i.e., that `repTree` is *precise* [15]). However, this is complicated by the fact that the `ptree` type is indexed by the height. The pure `treeSorted` predicate combines all of the facts about the key constraints, but is not necessary for the iterator and was explained in previous work [12], so we do not explain it in detail.

With our representation for B+ trees, we can now turn to their iterators. Our approach is similar to the technique we applied to the list iterator. First, we state the heap predicate that divides the tree into the “trunk” and the branches as disjoint entities. We can achieve this with only minor discomfort by parameterizing `repTree` by the leaf case and passing the empty heap when we only want to describe the trunk. We also implement a function `repLeaves` to describe a list of leaves in isolation. These two functions satisfy the following property which is key to opening and closing our iterator:

$$\begin{aligned} \forall h \text{ optr } p. \text{repTree } \text{optr } p &\iff \\ \text{repTrunk } \text{optr } p * \text{repLeaves } (\text{Some } (\text{firstPtr } p)) (\text{leaves } p) \text{ optr} & \end{aligned}$$

Using these predicates, we can define the representation of the iterator:

$$\begin{aligned} \text{repIter } \text{own } h \text{ m } \text{id}x &\stackrel{\Delta}{\iff} \exists h. \exists tr : \text{ptree } h. \exists i. \exists prev. \exists cur. \exists rest. \\ \text{own} &\mapsto (\text{ptrFor } tr, \#tr\#) * \text{repTrunk } h \text{ None } p * \\ &[m = \text{as_map } p] * [\text{treeSorted } h p \text{ MinMax}] * \\ h &\mapsto (cur, i) * \text{repLeaves } prev \text{ cur} * \text{repLeaves } rest \text{ None} * \\ &[\text{leaves } tr = prev ++ rest] * [\text{posInv } i \text{ id}x \text{ prev } rest \text{ m}] \end{aligned}$$

The first two lines after the existentials corresponds to the framed heap and pure facts needed to re-establish the tree representation invariant. The third line declares the iterator state ($h \mapsto (cur, i)$) and the combined `repLeaves` specify the representation of the leaves. Because each leaf could have a different number of key-value pairs, it is difficult to use the built-in `firstn` and `skipn` functions, so we existentially quantify two lists of leaves (`prev` and `rest`) and assert that their concatenation (`++`) must be equal to the leaves of the tree. The final pure fact establishes the invariant on the `cur` and the index into the current leaf: if there are elements left to iterate, $i + \text{length}(\text{as_map } prev) = idx$ and i is a valid index in the list. Otherwise, $i = 0$ and `rest = nil`.

6 Discussion

In this section we consider the overhead of verification (Section 6.1), summarize our sharing insights (Section 6.2), and review related work (Section 6.3).

6.1 The Burden of Mechanized Proofs

Our methodology places the burden of proof on the developer. Proof search scripts and lemmas are part of the final code and running them considerably increases compilation time. However, our proofs confirm strong functional correctness properties and our specifications document precise pre- and post-conditions for clients to use.

Figure 8, presents a quantitative look at the size of our development in number of lines. The *Spec* column counts command specifications; this is the interface that the client needs to reason about. Excluding the data structure invariants, this is the part of the code that a client of the library needs to reason about. The *Impl* column counts imperative code. The next two columns count auxiliary lemmas and automation. The second, *Sep. Lemmas*, counts lines that pertain to separation logic, while *Log. Lemmas* counts lines that only reason about pure structures, such as lists. The *Overhead* column gives the ratio of proofs to specification and code. The *Time* column gives the time required to prove all of the verification conditions not including auxiliary lemmas. Line counts include only new lines needed for verifying the function, so, if a lemma is required for both `sub` and `insert` it is only counted against `sub`.

As Figure 8 shows, the first commands contribute the most to the proof burden because we are writing general lemmas about the model and representation predicate. Once these lemmas have been proven, the remainder of the commands are almost immediate. We believe that the logical lemmas required for our code are mostly within the capabilities of existing automated theorem provers [13] and integrating such tools would likely eliminate all of the overhead from this column. It is less likely that existing tools are directly applicable to our separation logic though existing automation is fairly good at this. The time spent interactively verifying our implementation was mostly spent abstracting lemmas which is straightforward but time consuming because of Coq’s toplevel model.

Command	Lines of Code				Overhead	
	Spec	Impl	Log. Lemmas	Sep. Lemmas	Lines	Time (m:s)
<code>new</code>	2	1	1	15	5.33x	0:00
<code>free</code>	3	13	0	33	2.06x	0:15
<code>insert</code>	9	25	11	15	0.76x	1:22
<code>delete</code>	9	26	7	1	0.23x	2:52
<code>sub</code>	3	14	0	1	0.06x	1:21
<code>mfold_left</code>	7	13	6	1	0.35x	1:47
<code>iterator</code>	3	3	0	29	4.83x	0:17
<code>close</code>	3	2	0	9	1.80x	0:11
<code>next</code>	3	8	13	30	3.91x	2:30
Total	82	123	73	155	1.11x	12:07

Fig. 8. Breakdown of lines of code for lists and iterators.

6.2 Sharing Lessons

While originally proposed for parallel code, fractional permissions for external sharing are important for sequential code. This is a by-product of multiple views of the same data structure, in our case lists and iterators. Our solution is simple because the list and iterator are completely decoupled and so we do not need to correlate mutation through multiple views². Supporting mutation with a single iterator is relatively straight-forward though we need to change our iterator to carry the pointer to the list representation so that we can update the head pointer. The `ConcurrentModificationException` problem from Java is a general consequence of mutation of structures with multiple views over them and giving natural semantics to these operations is similar to the difficulty of writing precise specifications for concurrent functions.

When describing internal sharing, we get to specify equations directly on pointers. The difficulties come from scoping the existential quantification of pointers in recursive representation predicates. We find that quantifying all of the pointers at the beginning is useful for addressing this problem and it fits well with our solution to the problem of heap structures being loosely related to interface models because we can store the pointers in the interface model and easily encode aliasing. This approach also allows us separately to state pure facts about the structure of the heap rather than having to fold them into the representation predicate.

6.3 Related Work

Weide [20] uses *model-oriented specification* in *Resolve* to specify how iterators behave. These specifications follow a *requires/ensures* template on top of a purely logical model, similar to Ynot’s interface model.

² It should be noted that the code for maintaining multiple views is non-trivial, so the verification should not be expected to be trivial either.

Bierhoff [2] proposed a technique for using *type-state* specifications [10] for iterators. This system uses finite state machines to define the state of an object and specify when operations are permitted. This technique is particularly useful for specifying “non-interference” properties [19] such as marking a collection read-only when an iterator exists. We achieve this using fractional permissions, but can encode the same functionality by adding a state parameter to the representation predicate of our data structures.

Our approach is most similar to the work of Krishnaswami [11] where separation and Hoare logic are combined to reason about iterators. His technique relies on the separating implication ($-*$), the separation logic analog of implication. We are interested in incorporating this into our separation logic, but we have not yet developed effective automation for it, so the burden of using it can be considerable. More recent work by Jensen [9] shows how a similar approach using separating implication can be applied to mutable views of a container.

B+ trees have been formalized in two previous developments. Bornat *et al.* [3] proposed using classical conjunction to capture the B+ tree as a tree and a list in the same heap. This is convenient for representation, but it requires re-establishing both the heap as a tree and as a list at every step of the code. By unifying the two views, we only need to reason about the view that we are using in our code. We support the two views by proving `repTree` is equivalent to a representation that exposes the leaves as a list.

Sexton and Thielecke [17] formulate B+ trees by defining a language of tree-operations for a stack-machine. Their representation is similar to our own in not using classical conjunction, but they quantify structure in the representation predicate which forces them to state the pure properties there as well.

7 Conclusions

In this work we have demonstrated a technique for building verified imperative software using theorem proving in the Ynot library for Coq.

We showed how external sharing can be achieved using abstract predicates which quantify over fractional permissions and showed how this technique can be applied to representing multiple views. Further, we showed how ownership types can be applied to make the view’s representation predicate precise.

To address internal sharing we suggest simplifying recursive definitions by existentially quantifying all of the salient aspects of the data structure at the beginning of the representation predicate. This makes stating facts such as aliasing equations simple and allows the programmer to implement his or her code to minimize the use of existential quantification which can be difficult for automation to reason about.

Future Work

The use of the separating implication in so many developments [11,17] demonstrates its usefulness. It would benefit our own development by allowing us to

avoid duplicating parts of the representation predicate in the iterators. We are interested in extending Ynot's automation to reason about it and hope that doing so will reduce the burden of specifying and verifying Ynot code.

References

1. Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, et al. Mechanized Metatheory for the Masses: The PoplMark Challenge. 3603:50–65, 2005.
2. Kevin Bierhoff. Iterator specification with tpestates. In *SAVCBS '06*, pages 79–82, New York, NY, USA, 2006. ACM.
3. R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. *Proceedings of SPACE*, 4, 2004.
4. Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09*, pages 79–90, New York, NY, USA, 2009. ACM.
5. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64, New York, NY, USA, 1998. ACM.
6. Christian Haack and Clément Hurlin. Separation Logic Contracts for a Java-Like Language with Fork/Join. In *AMAST 2008*, pages 199–215, Berlin, Heidelberg, 2008. Springer-Verlag.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
8. John Boyland. Checking Interference with Fractional Permissions. 2694, 2003.
9. Braband Jensen Jonas. Specification and validation of data structures using separation logic. Master's thesis, Technical University of Denmark, 2010.
10. Taekgoo Kim, Kevin Bierhoff, Jonathan Aldrich, and Sungwon Kang. Typestate protocol specification in JML. In *SAVCBS '09*, pages 11–18, New York, NY, USA, 2009. ACM.
11. Neelakantan R. Krishnaswami. Reasoning about iterators with separation logic. In *SAVCBS '06*, pages 83–86, New York, NY, USA, 2006. ACM.
12. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *POPL'10*, January 2010.
13. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *In TACAS '08*, 2008.
14. Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP '06*, pages 62–73, New York, NY, USA, 2006. ACM.
15. Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.
16. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science, LICS'02*, 2002.
17. Alan Sexton and Hayo Thielecke. Reasoning about B+ Trees with Operational Semantics and Separation Logic. *Electronic Notes Theoretical Computer Science*, 218:355–369, 2008.
18. Matthieu Sozeau and Nicolas Oury. First-class type classes. 5170:278–293, 2008.
19. B. W. Weide, S. H. Edwards, D. E. Harms, and D. A. Lamb. Design and Specification of Iterators Using the Swapping Paradigm. *IEEE Trans. Softw. Eng.*, 20(8):631–643, 1994.
20. Bruce W. Weide. SAVCBS 2006 challenge: specification of iterators. In *SAVCBS '06*, pages 75–77, New York, NY, USA, 2006. ACM.