Mechanized Verification with Sharing

Gregory Malecha and Greg Morrisett

Harvard University SEAS

Abstract. We consider software verification of imperative programs by theorem proving in higher-order separation logic. Separation logic is quite effective for reasoning about tree-like data structures, but less so for data structures with more complex sharing patterns. This problem is compounded by some higher-order patterns, such as stateful iterators and visitors, where multiple clients need to share references into a data structure. We show how both styles of sharing can be achieved without sacrificing 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 as transformers of predicates over the state of the system. Unfortunately, stating and reasoning about these predicates is complicated in the presence of shared, mutable pointers. It was not until Reynolds proposed separation logic [16] that reasoning about pointer-based imperative programs in a modular way became tractable. However, even with this logic some verification tasks are far from simple, particularly when we cannot easily describe which abstractions "own" pointers, due to sharing.

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 to make algorithm and data structure implementations more efficient. Vanilla separation logic provides the first, but makes the second difficult because of non-local effects as illustrated by the following Java program:

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

This method creates an iterator object referencing the beginning of a list. Successive calls to the `next()` method are intended to step through the list. However, the call on line 3 removes the element that the iterator references, destroying the iterator's view on the list, even though line 3 does not mention the iterator. If problems like this go undetected at run-time, they can result in

A. Cavalcanti et al. (Eds.): ICTAC 2010, LNCS 6255, pp. 245–259, 2010.

`NullPointerExceptions` in Java, or memory corruption or segmentation faults in lower level languages such as C.

Separation logic can be used to reason about such programs, because it incorporates a notion of ownership: a command is allowed to access memory only if it appears in its "footprint" (i.e., pre- or post-condition). This gives us a frame property that ensures properties on sub-heaps that are *disjoint* from the footprint of the command are preserved, giving us an effective way to reason about abstracted, compound commands (i.e., method calls). However, in the example above, there is no way to give the iterator ownership over the list and still make the direct modification to the list at line 3. Rather, the iterator abstraction and the client must share the list. Consequently, after the state modification on line 3, we can no longer be ensured that the iterator's internal invariants hold, invalidating the call to `next()`.

Unfortunately, as originally conceived, separation logic is too restrictive to validate many patterns where an iterator or visitor safely shares state with the client or other abstractions. For example, if we called `lst.length()` on line 3, then the internal invariants of the iterator remain preserved, and we can safely call the `next()` method. But traditional separation logic does not support this kind of reasoning without exposing implementation details of the abstraction.

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.

We consider sharing of two sorts, *external* and *internal*. In *external* sharing, we wish to support multiple, simultaneous views of the same underlying memory, as in the iterator example above. In *internal* sharing, we wish to completely hide the sharing from the client so he/she can reason with a simple interface while the implementation is free to use aliasing for correctness or efficiency.

Contributions

We begin with a brief overview of the Ynot verification library [4] (Section 2), which provides an effective tool for writing and reasoning about higher-order, imperative programs with a type system that incorporates a form of separation-style, Hoare logic. We then:

- Show how fractional permissions [8] can be applied to provide sharing of high-level abstractions, focusing on collections. (Section 3)
- Show how external sharing can be leveraged to mechanically verify higher-order, effectful computations in Ynot, focusing on iterators. (Section 4)
- Show how internal sharing can be expressed by describing the representation of B+ trees, and how our approach simplifies the implementation of an iterator for traversing the leaves of the tree. (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.

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

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

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).

The empty heap (`emp`) denotes a heap containing no allocated cells (i.e., all pointers are mapped to `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 addressed by the pointer p. When $q = 1$, the points-to assertion grants code the ability to read, write, and deallocate the cell. When $q < 1$, the points-to relation gives read-only access to the heap cell. Informally, the separating conjunction ($*$) states that the two conjuncts hold on two "disjoint" pieces of the heap (denoted $h_0 \perp h_1$.) In the presence of 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 the merger of disjoint heaps. The

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

$\boxed{h \models P}$ Heap Propositions (*hprop*)

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

$\boxed{h_0 \perp h_1}$ Heap Disjointness

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

$\boxed{h_0 \uplus h_1}$ Heap Union

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

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

$$\begin{array}{ll}
\texttt{new} & : \Pi(T : \texttt{Type})(v : T), \texttt{Cmd}(emp)(p : ptr \Rightarrow p \mapsto v) \\
\texttt{free} & : \Pi(p : ptr), \texttt{Cmd}(\exists T, \exists v : T, p \mapsto v)(_ : unit \Rightarrow emp) \\
\texttt{read} & : \Pi(T : \texttt{Type})(p : ptr)(P : T \rightarrow hprop), \\
 & \quad \texttt{Cmd}(\exists v : T, p \overset{q}{\mapsto} v * P\ v)(v : T \Rightarrow p \overset{q}{\mapsto} v * P\ v) \\
\texttt{write} & : \Pi(T : \texttt{Type})(p : ptr)(v : T), \texttt{Cmd}(\exists T, \exists v' : T, p \mapsto v')(_ : unit \Rightarrow p \mapsto v) \\
\texttt{bind} & : \Pi(T\,U : \texttt{Type})(P\,P' : hprop)(Q : T \rightarrow hprop)(Q : U \rightarrow hprop), \\
 & \quad (\forall v : T, Q\,v \Longrightarrow P') \rightarrow \texttt{Cmd}(P)(Q) \rightarrow (T \rightarrow \texttt{Cmd}(P')(Q')) \rightarrow \texttt{Cmd}(P)(Q') \\
\texttt{return} & : \Pi(T : \texttt{Type})(v : T), \texttt{Cmd}(\texttt{emp})(r : T \Rightarrow [r = v]) \\
\texttt{cast} & : \Pi(T : \texttt{Type})(P\,P' : hprop)(Q\,Q' : T \rightarrow hprop), (P' \Longrightarrow P) \rightarrow \\
 & \quad (\forall v : T.Q\ v \Longrightarrow Q'\ v) \rightarrow \texttt{Cmd}(P)(v : T \Rightarrow Q\ v) \rightarrow \texttt{Cmd}(P')(v : T.Q'\ v) \\
\texttt{frame} & : \Pi(T : \texttt{Type})(P\,Q\,R : hprop), \texttt{Cmd}(P)(r : T)(Q\,r) \rightarrow \\
 & \quad \texttt{Cmd}(P * R)(r : T \Rightarrow Q\,r * R)
\end{array}$$

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

separation logic of Ynot also supports existential quantification, and injection of pure propositions (propositions that do not mention the heap such as $n < 5$).

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, and by the

frame property, the resulting pointer must be fresh (disjoint from the rest of the heap). 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 $\mathrm{P} = \mathbf{fun}\ r \Rightarrow r \mapsto v$ thus making the post-condition reduce to $p \overset{q}{\mapsto} r * r \mapsto v$ which maintains the connection between p and 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 foundations 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 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 described as a predicate over heaps (*hprop*). 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 $t \overset{q}{\mapsto} l$. 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.

```
Fixpoint specNth {T} (ls : list T) (n : nat) : option T :=
  match ls, n with
    | nil , _       ⇒ None
    | a :: _, 0     ⇒ Some a
    | _ :: b, S n ⇒ specNth b n
  end
Class ImpList (T : Type) (tlst : Type) := {
  (* tlst is a handle to perm capabilities to the list T *)
   llist  : perm → tlst → list T → hprop ;
  (* Fractional merging and splitting of lists *)
   llist_split  : ∀ q q' t m, q |#| q' →
     llist (q + q') t m ⟺ llist q t m ∗ llist q' t m ;
  (* Allocate an empty list *)
  new : Cmd (emp) (res : tlst ⇒ llist 1 res nil) ;
  (* Free the list *)
  free : Π (t : tlst),
    Cmd (∃ ls : list T, llist 1 t ls) (_ : unit ⇒ emp) ;
  (* Get the ith element from the list if it exists. *)
  sub : Π (t : tlst T) (i : nat) (m : #list T#) (q : #perm#),
    Cmd (llist q t m)
        (res : option T ⇒ llist q t m ∗ [res = specNth m i]);
  (* Insert an element at the ith position in the list. *)
  insert : Π (t : tlst) (v : T) (i : nat) (m : #list T#),
    Cmd (llist 1 t m)
        (_ : unit ⇒ llist 1 t (specInsert v i m))
}
```

Fig. 3. Externally-sharable list interface

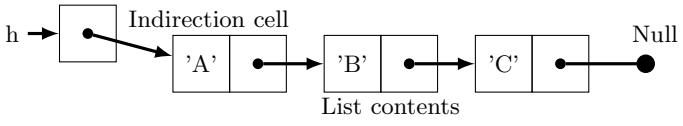

Fig. 4. A heap representing the list `['A', 'B', 'C']`

$$\text{llseg } q \; from \; to \; \texttt{nil} \overset{\Delta}{\iff} [from = to] \tag{1}$$

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

$$\text{tlst } T \;=\; \text{ptr} \tag{3}$$

$$\text{llist } q \; t \; ls \overset{\Delta}{\iff} \exists hd.\; t \overset{q}{\mapsto} hd * \text{llseg } hd \; \text{Null } ls \tag{4}$$

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 ($from \overset{q}{\mapsto} \text{mkNode } a \; x$) and x points to the rest of the list

(llseg q x *to* b). Equations (3) and (4) make 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 llist_split lemma that states a $q + q'$ fraction of the list is equivalent to a q fraction of the list disjoint from a q' 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 and efficient iterator.

We define the iterator with a representation predicate (liter) and commands for creating (open), advancing (next), and deallocating (close) it:

```
1  Parameter titr : Type → Type.
2  Parameter liter : ∀ T. perm → tlst T →
3    titr T → list T → nat → hprop.
4  Parameter open : Π (T : Type) (t : tlst T) (m : #list T#)
5    (q : #perm#),
6    Cmd (llist q t m) (res : titr T ⇒ liter q t res m 0).
7  Parameter next : Π (T : Type) (t : titr T) (m : #list T#)
8    (idx : #nat#) (own : #tlst T#) (q : #perm#),
9    Cmd (liter q own t m idx)
10       (res : option T ⇒ liter q own t m (idx + 1) ∗
11          [res = specNth m idx]).
12 Parameter close : Π (T : Type) (t : titr T) (own : #tlst T#)
13   (m : #list T#) (q : #perm#),
14   Cmd (∃ idx, liter q own t m idx) (_ : unit ⇒ llist q own m).
```

The heap proposition liter q l i m n describes the iterator i to the n^{th} element of the imperative list l which is a representation of the functional list m. Here, the fractional permission q is the ownership of the underlying list, not of the iterator, so even if it is not 1, we will be able to modify the iterator, just not the underlying list. The open computation constructs an iterator to the beginning of a `tlst T` by converting the heap predicate from `llist 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 llist .

The own parameter specifies the owner of the iterator in the style of ownership types [5]. We make it a parameter so that we can specify the invariant precisely enough to prove that the post-condition to close is exactly the same as the pre-condition to open so that use of an iterator loses no information about the underlying list.

In this interface, an iterator requires 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, 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.

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

5 Internal Sharing and 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 technique, 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.

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, deallocation, key lookup, and key-value insertion. The iterator interface is the same as the list iterator interface except it does not have fractional permissions.

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 key-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 is that the model does not totally determine the heap structure. The second is the, potentially non-local, pointer aliasing in the leaves.

```
(* tfmap is the type of finite maps from key to value. *)
Class FiniteMap (K V : Type) (tfmap : Type) := {
  (* The tfmap handle represents the fmap. *)
  repMap : tfmap → fmap K V → hprop ;
  new : Cmd emp (h : tfmap ⇒ repMap h nil) ;
  insert : Π (h : tfmap) (k : K) (v : V) (m : #fmap K V#),
    Cmd (repMap t m)
        (res : option V ⇒ repMap h (specInsert v m) *
            [res = specLookup k m]) ;
  (** ... free & lookup ... **)
  (** The iterator **)
  titr : Type ;
  repIter : tfmap → titr → fmap K V → nat → hprop ;
  open : Π (h : tfmap) (m : #fmap K V#),
    Cmd (repMap h m) (res : titr ⇒ repIter h res m 0) ;
  next : Π (h : titr) (own : #tfmap#) (m : #fmap K V#)
    (idx : #nat#),
    Cmd (repIter own h m idx)
        (res : option (K * V) ⇒
            repIter own h m (idx + 1) * [specNth m idx]) ;
  close : Π (h : titr) (own : #tfmap#) (m : #fmap K V#),
    Cmd (∃i. repIter own h m i) (_ : unit ⇒ repMap own m)
}
```

Fig. 5. The imperative finite map interface

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 we must witness these existentials in every place that we deconstruct the tree. 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:

```
Fixpoint ptree (h : nat) : Type :=
  match h with
    | 0    ⇒ list (key * value)
    | 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. However, when the rightmost leaf of one tree must alias the leftmost leaf of the next the pointers have disjoint scopes making it impossible to

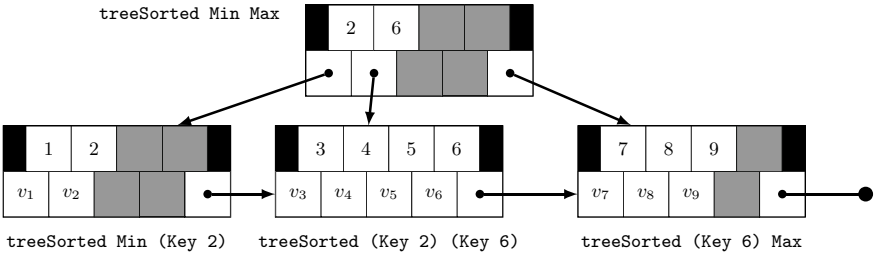

Fig. 6. A B+ tree of arity 4 ($n = 4$) for the finite map from $i \mapsto v_i$ for $1 \leq i \leq 9$

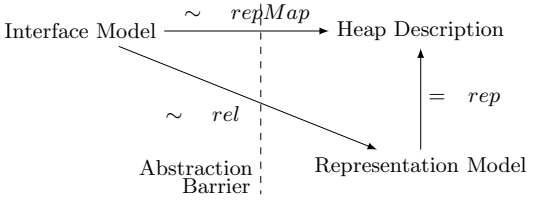

Fig. 7. Decouple the relational mapping between the interface and the heap by factoring out a representation model that is functionally related to the heap

relate them directly. Changing the representation to quantify the pointers at the lowest ancestor of the two nodes complicates the recursion because we have to handle the first subtree specially. This strategy also leads to difficulties when describing iterators because we will want to ignore the "trunk" 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 :=
2   ptr * match h with
3           | 0    ⇒ list (key * value)
4           | S h' ⇒ list (key * ptree h') * ptree h'
5         end
```

Using this representation model, we can easily describe the aliasing without worrying about scoping since all of the pointers are quantified at the root.

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

$$\texttt{repTree}\ 0\ optr\ (p', ls) \overset{\Delta}{\Longleftrightarrow} \exists ary.\ p' \mapsto \texttt{mkNode}\,0\ ary\ optr\ *\ \texttt{repLeaf}\ ary\ ls$$

$$\texttt{repTree}\ (1+h)\ optr\ (p', (ls, nxt)) \overset{\Delta}{\Longleftrightarrow}$$
$$\exists ary.\ p' \mapsto \texttt{mkNode}\ (h+1)\ ary\ (\texttt{ptrFor}\ nxt) *$$
$$\texttt{repBranch}\ ary\ (\texttt{firstPtr}\ nxt)\ ls\ *\ \texttt{repTree}\ h\ optr\ nxt$$

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

$$\texttt{repLeaf}\ ary\ [v_1, \ldots, v_m] \stackrel{\Delta}{\Longleftrightarrow} \\ ary[0] \mapsto \texttt{Some}\ v_1 * \cdots * ary[m-1] \mapsto \texttt{Some}\ v_m * \\ ary[m] \mapsto \texttt{None} * \cdots * ary[n-1] \mapsto \texttt{None}$$

In the branch case, the array holds key-pointer pairs such that each pointer points to the corresponding subtree. This is captured by the `repBranch` predicate. Note that we use `ptrFor` to *compute* the pointers from the model.

$$\texttt{repBranch}\ ary\ optr\ [(k_1, t_1), \ldots, (k_m, t_m)] \stackrel{\Delta}{\Longleftrightarrow} \\ ary[0] \mapsto \texttt{Some}\ (k_1, \texttt{ptrFor}\ t_1) * \texttt{repTree}\ h\ (\texttt{firstPtr}\ t_2)\ t_1\ * \cdots * \\ ary[m-1] \mapsto \texttt{Some}\ (k_m, \texttt{ptrFor}\ t_m) * \texttt{repTree}\ h\ optr\ t_m\ * \\ ary[m] \mapsto \texttt{None} * \cdots * ary[n-1] \mapsto \texttt{None}$$

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 associated with the tree (using as_map) and stating that it equals the desired model. We can then pick the handle type to be a pointer and define the full representation predicate to be the conjunction of *rep*, *rel* and the pure facts about the tree structure.

$$\texttt{repMap}\ hdl\ m \stackrel{\Delta}{\Longleftrightarrow} \exists h. \exists p : \texttt{ptree}\ h. hdl \mapsto (\texttt{ptrFor}\ p, \#p\#)\ * \\ \texttt{repTree}\ h\ \texttt{None}\ p\ *\ [m = \texttt{as\_map}\ p]\ *\ [\texttt{treeSorted}\ h\ p\ \texttt{Min}\ \texttt{Max}]$$

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:

$$\forall h\ optr\ p.\ \texttt{repTree}\ optr\ p \Longleftrightarrow \\ \texttt{repTrunk}\ optr\ p * \texttt{repLeaves}\ (\texttt{Some}\ (\texttt{firstPtr}\ p))\ (\texttt{leaves}\ p)\ optr$$

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

$$\begin{array}{l}\texttt{repIter}\ own\ h\ m\ idx \stackrel{\Delta}{\iff} \exists h.\exists tr : \texttt{ptree}\ h.\exists i.\exists prev.\exists cur.\exists rest.\\ \quad own \mapsto (\texttt{ptrFor}\ tr, \#tr\#) * \texttt{repTrunk}\ h\ \texttt{None}\ p\ *\\ \quad [m = \texttt{as\_map}\ p] * [\texttt{treeSorted}\ h\ p\ \texttt{Min}\ \texttt{Max}]\ *\\ \quad h \mapsto (cur, i) * \texttt{repLeaves}\ prev\ cur * \texttt{repLeaves}\ rest\ \texttt{None}\ *\\ \quad [\texttt{leaves}\ tr = prev\ +\!\!+\ rest] * [\texttt{posInv}\ i\ idx\ prev\ rest\ m]\end{array}$$

The first two lines after the existentials correspond 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 *cur* and the index into the current leaf: if there are elements left to iterate, $i + \texttt{length}\,(\texttt{as\_map}\,prev) = idx$ and i is a valid index in the list. Otherwise, $i = 0$ and $rest = \texttt{nil}$.

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 functional (partial) 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 first, *Sep. Lemmas*, counts lines that pertain to separation logic, while *Log. Lemmas* counts lines that only reason about pure structures, e.g. 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

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

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

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 interaction model.

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 views[2]. Supporting mutation with a single iterator is relatively straightforward. The `ConcurrentModificationException` problem from Java is a general consequence of mutation of structures with multiple views and giving natural semantics to these operations is similar to the difficulty of writing precise specifications for concurrent functions.

When describing internal sharing, our technique allows us to specify equations directly on pointers. We find that quantifying all of the pointers at the beginning is useful for addressing this problem and it integrates nicely with our solution to heap structures being loosely related to interface models.This also allows us separately to state pure facts about the shape rather than having to fold them into the representation predicate. It is unclear how this technique could be applied to concurrent code, however, since this irrelevant, global state would need to be protected by a lock.

[2] The code for maintaining multiple concurrently mutable views is not simple, so the verification should not be expected to be trivial either.

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.

Bierhoff [2] proposed 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 type states 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 one view at a time. 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 minimize the use of existential quantification which can be difficult to reason about automatically.

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. Aydemir, B.E., Bohannon, A., Fairbairn, M., et al.: Mechanized Metatheory for the Masses: The PoplMark Challenge. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS, vol. 3603, pp. 50–65. Springer, Heidelberg (2005)
2. Bierhoff, K.: Iterator specification with typestates. In: SAVCBS 2006, pp. 79–82. ACM, New York (2006)
3. Bornat, R., Calcagno, C., OHearn, P.: Local reasoning, separation and aliasing. In: Proceedings of SPACE, vol. 4 (2004)
4. Chlipala, A., et al.: Effective interactive proofs for higher-order imperative programs. In: ICFP 2009, pp. 79–90. ACM, New York (2009)
5. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998, pp. 48–64. ACM, New York (1998)
6. Haack, C., Hurlin, C.: Separation Logic Contracts for a Java-Like Language with Fork/Join. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 199–215. Springer, Heidelberg (2008)
7. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)
8. Boyland, J.: Checking Interference with Fractional Permissions, vol. 2694 (2003)
9. Jonas, B.J.: Specification and validation of data structures using separation logic. Master's thesis, Technical University of Denmark (2010)
10. Kim, T., Bierhoff, K., Aldrich, J., Kang, S.: Typestate protocol specification in JML. In: SAVCBS 2009. ACM, New York (2009)
11. Krishnaswami, N.R.: Reasoning about iterators with separation logic. In: SAVCBS 2006, pp. 83–86. ACM, New York (2006)
12. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: POPL 2010 (January 2010)
13. De Moura, L., Bjrner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
14. Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In: ICFP 2006. ACM, New York (2006)
15. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science 375(1-3), 271–307 (2007)
16. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Symposium on Logic in Computer Science, LICS 2002 (2002)
17. Sexton, A., Thielecke, H.: Reasoning about B+ Trees with Operational Semantics and Separation Logic. Electronic Notes in Theoretical Computer Science 218, 355–369 (2008)
18. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
19. Weide, B.W., et al.: Design and Specification of Iterators Using the Swapping Paradigm. IEEE Trans. Softw. Eng. 20(8), 631–643 (1994)
20. Weide, B.W.: SAVCBS 2006 challenge: specification of iterators. In: SAVCBS 2006, pp. 75–77. ACM, New York (2006)