Reasoning About Iterators With Separation Logic

Neelakantan R. Krishnaswami
Carnegie Mellon University
neelk@cs.cmu.edu

ABSTRACT

Separation logic is an extension of Hoare logic which permits reasoning about imperative programs that use shared mutable heap structure. In this note, we show how to use higher-order separation logic to reason abstractly about an iterator protocol.

Categories and Subject Descriptors

D.2 [**Software/Program Verification**]: Correctness Proofs

General Terms

Languages, Verification

Keywords

separation logic, iterators, aliasing, challenge problem

1. JAVA STYLE ITERATORS

The iterator interface[3] in Java works roughly as follows. First, we have a mutable collection type. This type supports a number of operations, some of which like `add`-ing an element to a collection will mutate the collection, and others, like checking to see if it is `empty`, which do not modify the collection.

To get the elements of a collection, we create another mutable object called an iterator. This object has a method `next`, which returns a new element of the collection each time it is called, finally failing when there are no more elements within it.

However, both the collection and the iterator are imperative objects, and correct usage of an iterator also requires observing some additional restrictions to ensure that the state of an iterator and its underlying collection remain in sync. Specifically, a client program:

Fifth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), November 10–11, 2006, Portland, Oregon, USA.

- may create as many iterators on a single collection as they like,
- may freely call any methods on the collection that do not change the collection's observable state (such as `empty`),
- may freely call `next` on the iterators in any order, and
- may **NOT** call `next` on an iterator after calling `add` on the underlying collection.

The general idea is that an iterator maintains a pointer into some part of the collection during its traversal, and that updating the collection may cause the iterator's reference to point to an incorrect part of the collection. So while an iterator is live, dangerous method calls to its underlying collection should be forbidden, and only safe method calls permitted.

2. SEPARATION LOGIC

Separation logic [5, 6] is an extension of Hoare logic [4] intended to simplify reasoning about aliasing with mutable data structures. We have developed a version of separation logic that permits reasoning about imperative programs in high level languages (such as Java or ML), and which uses features of higher-order logic [2] to reason abstractly about high-level aliasing behavior.

Separation logic extends the logical language of preconditions and postconditions with two new logical connectives, the separating conjunction "$A * B$", and the magic wand "$A -\!\!* B$". Intuitively, $A * B$ means that A holds in one part of the heap, and B holds in a disjoint part of the heap. This contrasts with the regular conjunction $A \wedge B$, which means that both A and B hold in the current heap. The magic wand $A -\!\!* B$ means that if you added a piece of storage which validated A to the current heap, the whole thing would validate B. (Likewise, the informal meaning of $A \supset B$ is that if A holds in the current heap, then B will too.) Finally, the separating and ordinary connectives can be freely mixed, which lets us describe quite complex aliasing behaviors.

We use the propositions of separation logic to describe the pre- and post-conditions of commands, and describe the behavior of commands with Hoare triples of the form $\{P\}\ c\ \{a : \tau.\ Q\}$. The P is the state the heap must be in before the command can be run, and Q describes the changed heap after the command finishes. Since side-effecting operations can also return values, we use the $a : \tau$ notation to name the return value in the postcondition.

$\exists$`new_coll, size, add, new_iter, next`.
$\exists coll$: ((ref list $\times$ ref $\mathbb{N}$) $\times$ seq nat $\times$ prop) $\Rightarrow$ prop.
$\exists iter$: (ref list $\times$ (ref list $\times$ ref $\mathbb{N}$) $\times$ seq nat $\times$ prop) $\Rightarrow$ prop.

$\{\top\}$ `new_coll()` $\{a : (\text{ref list} \times \text{ref } \mathbb{N}).\ \exists P.\ coll(a, [], P)\}$ and

$\forall P, c, x, xs.\ \{coll(c, xs, P)\}$
`add`(c, x)
$\{a : 1.\ \exists P'.\ coll(c, x :: xs, P')\}$ and

$\forall P, c, xs.\ \{coll(c, xs, P)\}$
`empty`(c)
$\{a : \text{bool}.\ coll(c, xs, P)\}$ and

$\forall c, xs, P.\ \{coll(c, xs, P)\}$
`new_iter`(c)
$\{a : \text{ref list}.\ iter(a, c, xs, P)\}$ and

$\forall i, c, xs, P.\ \{iter(i, c, xs, P)\}$
`next`(i)
$\{a : 1 + \text{nat}.\ iter(i, c, xs, P)\}$ and

$\forall i, c, xs, P.\ \{iter(i, c, xs, P) \supset coll(c, xs, P) *$
$coll(c, xs, P) -\!\!* iter(i, c, xs, P)\}$

Figure 1: Iterator Specification

Since a triple only specifies the behavior of a single routine, we combine triples into specifications, which are logical formulas that use triples as their atomic propositions. So we can specify an interface to a module by taking the conjunction of the triples for each operation, and then existentially quantifying over the implementations.

3. THE ITERATOR PROBLEM

3.1 Iterator Interface Specification

We give a concrete example of this idea in Figure 1, which is the specification for iterators. In this example, our overall collection type is the pair ref list $\times$ ref $\mathbb{N}$. The first field has type ref list, which is the type mutable linked lists of integers, and the second field is a pointer to a natural number. This field tracks the number of times a harmless method like `empty`, which helps illustrate the fact that the *concrete* state of an object can change even while its *abstract* state remains the same.

To describe the heap behavior, we introduce a pair of existentially quantified predicates, *coll* and *iter*. These predicates permit us to talk about the mutable state associated with collections and iterators, without revealing their concrete implementation. The predicate $coll(c, xs, P)$ asserts that the collection object c represents the abstract sequence xs, and that it is in an abstract state P. We represent abstract states using propositions, which is why we need higher-order logic. The assertion $iter(i, c, xs, P)$ asserts that i is an iterator over the collection c with elements xs and abstract state P.

For example, the specification

$$\{\top\}\ \texttt{new\_coll()}\ \{a : \tau_c.\ \exists P.\ coll(a, [], P)\}$$

states that starting from any heap, calling `new_coll` will return a new mutable list and heap structure corresponding to that list. The specification

$$\{coll(c, xs, P)\}\ \texttt{empty}(c)\ \{a : \text{bool}.\ coll(c, xs, P)\}$$

```
1   {coll(c, xs, P)}
2   letv b = empty(c) in
3   {coll(c, xs)}
4   letv i1 = new_iter(c) in
5   {iter(i1, c, xs, P)}
6   {(coll(c, xs, P) * (coll(c, xs, P) -* iter(i1, c, xs, P))}
7   letv i2 = new_iter(c) in
8   {iter(i2, c, xs, P) * (coll(c, xs, P) -* iter(i1, c, xs, P))}
9   {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
10  letv b' = empty(c) in
11  {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
12  {iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
13  letv v = next(i1) in
14  {iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
15  {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
16  {iter(i2, c, xs, P))*
      (coll(c, xs, P) -* iter(i1, c, xs, P))}
17  letv v = next(i2) in
18  {iter(i2, c, xs, P))*
      (coll(c, xs, P) -* iter(i1, c, xs, P))}
19  {coll(c, xs, P)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
20  letv _ = add(c, x) in
21  {∃Q. coll(c, xs, Q)*
      (coll(c, xs, P) -* iter(i1, c, xs, P))*
      (coll(c, xs, P) -* iter(i2, c, xs, P))}
```

Figure 2: Iterator Client

asserts that the `empty` function will return a boolean, and that it will leave the abstract state unchanged. Note that we could give a more precise specification (e.g., that the `empty` function returns true if the collection is empty and false otherwise). We choose not to in order to focus this example on aliasing.

By way of contrast, the specification for `add`

$$\{coll(c, xs, P)\}\ \texttt{add}(c, x)\ \{a : 1.\ \exists P'.\ coll(c, x :: xs, P')\}$$

says that adding an element to the collection will alter the abstract state of the object. We existentially quantify over the abstract state in the postcondition, to show we can no longer assume that P' is the same as P.

The specification for `new_iter`

$$\{coll(c, xs, P)\}\ \texttt{new\_iter}(c)\ \{a.\ iter(a, c, xs, P)\}$$

says that if we start with a collection c, then we can consume it to construct an iterator.

The `next` function has the specification

$$\{iter(i, c, xs, P)\}\ \texttt{next}(i)\ \{a : 1 + \text{nat}.\ iter(i, c, xs, P)\},$$

which says that if we have an iterator i, then `next`(i) will give us an integer or signal a failure. As with `empty`, we do not model the behavior of the iterator in any further detail — the spec could easily be refined further, but that detail would not be relevant to the issue of reasoning about aliasing. The detail that is relevant is the fact that the iterator preserves the abstract collection state P, which is

how we describe the fact that the iterator does not modify the underlying collection.

That said, a natural question is how we can create two iterators on the same collection, because the `new_iter` function transforms a $coll(c, xs)$ state to an $iter(i, c, xs, P)$ state, which means that the precondition to call `new_iter` no longer holds. This is where the *sharing axiom* comes into play – the final invariant in the specification:

$$iter(i, c, xs, P) \supset [coll(c, xs, P) * \\ coll(c, xs, P) -\!\!* iter(i, c, xs, P)]$$

is a separation logic formula that describes how to recover a collection from an iterator state. It says that if we have an iterator state $iter(i, c, xs, P)$, then that state can be viewed as two disjoint pieces, one of which is the original collection (with the invariant P maintained), and one piece that can be combined with the collection to restore the iterator.

The sharing axiom makes use of the fact we have both standard implication and separating implication available in the same logic. We use implication to reason that the same piece of state can be viewed in multiple ways, and the separating implication to reason about one isolated part of the state.

3.2 Iterator Client Usage

We can see an example of how a client would make use of this specification in Figure 2. On line 1, we see that the precondition for our program is that the variable c holds a collection. On line 4, we create an iterator i_1, consuming the collection to produce an iterator, as seen in the state on line 5. We now apply the sharing axiom on line 6 to break the iterator state into two pieces, which lets us create a second iterator bound to i_2.

The program state on line 8 contains an iterator for i_2, and some state that will let us reconstruct i_1's iterator. On line 9 we apply the sharing axiom once more, to break out the collection state again, and this lets us call `empty` on line 10.

On line 12, we use the collection and an i_1's iterator fragment to recover the precondition for calling `next`(i_1) on line 13, and then on lines 14-16, we apply the sharing axiom and combine the iterator state fragment for i_2, so that we can call `next`(i_2) on line 17.

The informal idea should be coming into focus now – we are transferring ownership of the collection between the different collections, using the sharing axiom to get a collection out of an iterator, and the deduction rule for magic wand ($A * (A -\!\!* B)$ entails B) to put it back in.

On line 18 and 19, we once again use the sharing axiom to disassemble the iterator and get back the collection, and then call `add`(c, x) on line 21. This gives us a state in which $\exists Q.\ coll(c, x :: xs, Q)$ holds. We can no longer apply the separating implication law to get a full iterator state, because we need a hypothesis of the form $coll(c, xs, P)$ to recover an *iter* state, and we don't know whether Q is the same as P. As a result, we can no longer call `next` on either i_1 or i_2 any longer, just as we desire.

So the Hoare triples and sharing axioms put us in a situation where we can create multiple iterators, and can freely call methods on the collection which don't change its abstract state, but which also enforce the property that there can be no calls to `next` after modifying the collection – and the client was able to do this without knowing anything about the details of the internal heap structure of the collection.

Interestingly, the abstract states in our spec are reminiscent of a consistency check the Java collection libraries perform. The Java libraries keep a sequence number for each collection, and update it when the collection is modified. Iterators save the sequence number when they are created, and will raise a runtime error if the underlying collection's sequence number ever differs from their saved value. With our specification, the abstract state changes whenever we call a dangerous method, and our (static) verification is kept from proceeding.

3.3 Iterator Implementation

Finally, in Figure 3, we give an example implementations for this specification. The specification is a big existential quantifier, and so our implementations are the witnesses to this existential type. For the abstract program variables (such as `next` or `empty`) we give function definitions. A collection is a linked list and a counter, and an iterator is a pointer to the interior of a list.

Most of these definitions manipulate imperative linked lists in the obvious way, but it's worth examining `empty`. A call to this function modifies the state of the collection, but in a safe way. It updates the counter, but does not modify the linked list, so iterators over the collection will not be invalidated. More elaborate examples might be something like a collection that does memoization or a splay tree that rebalances after each query. In each of these cases, the abstract state of the object does not change, even though its in- memory representation might.

We demonstrate this idea in the definitions of the existentially quantified predicates. These predicate definitions are given as functions of their input, which take in data and return propositions. The definition of the $coll(c, xs, P)$ predicate is an assertion that a collection value's first field points to an integer counter, and that its second field is a linked list representing xs, and is also in state P. The *linked_list* predicate is a recursive function on xs, which permits us to define an inductive predicate characterizing linked lists.

The *iter* predicate, for example, is an assertion stating that the iterator points to an interior pointer of the linked list, and that the predicate variable P is preserved for the whole list.

In this example, we have focused on being able to abstractly specify and reason about the imperative aspects of modules. Of course, one would also like the iterator specification to be abstract in the implementation types used for collections and iterators (here ref list $\times$ ref N), i.e., to have existential quantification over *types* to model abstract data types. We plan on addressing this in future work.

4. CONCLUSIONS

Theorem proving in higher-order logic has a long and notorious history of being very difficult to automate, and the addition of separation logic will not make this task any easier. However, different kinds of partial automation are probably feasible, and what follows are hopefully-educated, possibly-wild, guesses about the difficulty of different levels of automation.

The simplest level is just verifying that an annotated pro-

gram is actually correct with respect to our program logic. This should be quite straightforward to implement using a tactical theorem prover such as Coq or Isabelle, though we have not actually implemented this.

The next easiest task will be to automatically verify that a client program respects a given specification. The sorts of manipulations we performed in the sample client code did not make essential use of higher-order logic, since the predicate variables representing abstract states were never instantiated. Assuming this is a general pattern, checking client code should not require more know-how than checking programs that use first-order separation logic. This is still a fairly difficult problem, though substantial progress has been made with Smallfoot [1].

Automatically checking that implementations satisfy a given specification is almost certainly a much harder problem. We must construct functions that show how to realize abstract predicates (such as $coll(c, xs, P)$), and finding them can require real creativity. However, it *may* be possible to partially automate checking the function bodies given all the predicate definitions.

Finally, inferring specifications and module boundaries from programs seems completely out of reach, since the relevant abstraction boundaries simply are not evident in the code, and even skilled human programmers find identifying them a very difficult task.

However, the main purpose of this line of research is not to produce ready-to-use tools. Instead, we are trying to construct a very rich specification language capable of describing how aliasing is used as concisely and naturally as possible. Our hope is that having simple mathematical characterizations of the realistic aliasing patterns will make it easier to construct and validate more limited (and hence more automatable) methods that verify exactly and only the forms of aliasing used in well structured programs.

$$
\begin{array}{lll}
\texttt{new\_coll}\ () & \equiv & \mathsf{letv}\ counter = \mathsf{new}_{\mathbb{N}} 0\ \mathsf{in} \\
& & \mathsf{letv}\ list = \mathsf{new}_{\mathsf{list}} \mathsf{nil}\ \mathsf{in} \\
& & (list, counter)
\end{array}
$$

$$
\begin{array}{lll}
\texttt{add}(c, x) & \equiv & \mathsf{letv}\ cell = !(\mathsf{fst}\ c)\ \mathsf{in} \\
& & \mathsf{letv}\ t = \mathsf{new}_{\mathsf{list}} cell\ \mathsf{in} \\
& & c := \mathsf{cons}(x, t)
\end{array}
$$

$$
\begin{array}{lll}
\texttt{empty}(c) & \equiv & \mathsf{letv}\ cell = !(\mathsf{fst}\ c)\ \mathsf{in} \\
& & \mathsf{letv}\ \_ = \mathsf{increment}(\mathsf{snd}\ c)\ \mathsf{in} \\
& & \mathsf{listcase}(cell, \mathsf{true}, (h, t).\ \mathsf{false})
\end{array}
$$

$$
\texttt{new\_iter}(c) \equiv \mathsf{new}_{\mathsf{ref\ list}}(\mathsf{fst}\ c)
$$

$$
\begin{array}{lll}
\texttt{next}(i) & \equiv & \mathsf{letv}\ c = [!i]\ \mathsf{in} \\
& & \mathsf{letv}\ cell = [!c]\ \mathsf{in} \\
& & \mathsf{letv}\ ans = \mathsf{listcase}(cell, \mathsf{None}, \\
& & \qquad (h, t).\ \mathsf{letv}\ \_ = i := t\ \mathsf{in}\ \mathsf{Some}\ h)\ \ \mathsf{in} \\
& & ans
\end{array}
$$

$$
coll(c, xs, P) \equiv \exists n.\ \mathsf{snd}\ c \hookrightarrow n * (linked\_list(\mathsf{fst}\ c, xs) \wedge P)
$$

$$
\begin{array}{lll}
linked\_list(c, x :: xs) & \equiv & \exists c'.\ c \hookrightarrow \mathsf{cons}(x, c') * linked\_list(c', xs) \\
linked\_list(c, [\,]) & \equiv & c \hookrightarrow \mathsf{nil}
\end{array}
$$

$$
\begin{array}{lll}
seg(l, l', x :: xs) & \equiv & \exists l''.\ l \hookrightarrow \mathsf{cons}(x, l'') * seg(l'', xs) \\
seg(l, l', [\,]) & \equiv & l = l'
\end{array}
$$

$$
\begin{array}{lll}
iter(i, c, xs, P) & \equiv & \exists l, n, xs_1, xs_2. \\
& & \qquad (P \wedge (seg(\mathsf{fst}\ c, l, xs_1)\ * coll(l, xs_2))) * \\
& & \qquad i \hookrightarrow l\ * \mathsf{snd}\ c \hookrightarrow n \wedge \\
& & \qquad xs = xs_1 \cdot xs_2
\end{array}
$$

Figure 3: Iterator Implementation

5. REFERENCES

[1] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects*, Amsterdam, The Netherlands, 2001.

[2] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP 2005: The European Symposium on Programming*, pages 233–247, Edinburgh, Scotland, April 2005.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[4] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(583):576–580, 1969.

[5] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'01)*, London, 2001.

[6] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, Copenhagen, Denmark, July 2002. IEEE Press.