15-414: Bug Catching: Automated Program Verification

Lecture Notes on Programs with Arrays

Matt Fredrikson Ruben Martins

Carnegie Mellon University

Lecture 6

1 Introduction

The previous lecture focused on loops, starting with axioms and leading to a derived rule that allows us to simplify reasoning about loops to reasoning about the behavior of a single iteration of their bodies. We worked an example involving a program that uses loops to compute the square of a number, and found that much of the difficulty in reasoning about loops lies in finding a suitable invariant.

Loops are frequently used to compute over a sort of programming element that we haven't introduced yet, namely arrays. Arrays are an important data structure in imperative programming, enabling us to implement things that we couldn't otherwise. However, they introduce significant complexity into programs' behavior, so sound reasoning about their use is crucial. Today we will introduce arrays into our language, and use our semantics to argue the correctness of the binary search code we discussed in the first lecture. Along the way, we will build more experience in finding suitable loop invariants for increasingly complex code, and in all likelihood, learn to appreciate the benefits of using an SMT solver to complete our proofs.

2 Recall: Loop Invariants

Last lecture we derived the loop rule using the dynamic logic sequent calculus. We started from the unwind and unfold axioms,

$$([\text{unwind}])\quad [\mathtt{while}(Q)\,\alpha]P \leftrightarrow [\mathtt{if}(Q)\,\{\alpha;\mathtt{while}(Q)\,\alpha\}]P$$

$$([\text{unfold}])\quad [\mathtt{while}(Q)\,\alpha]P \leftrightarrow (Q \rightarrow [\alpha][\mathtt{while}(Q)\,\alpha]P) \land (\neg Q \rightarrow P)$$

The issue with using these axioms to reason about a program is that they do not make our work easier: in either case, they reduce reasoning about a while loop to a bit of logic, and then more reasoning about exactly the same while loop. In principle, as long as we knew how many times a loop would execute in advance, we could use these axioms to unfold the loop body until the condition implies termination. But we will not in general know this information, so we needed different ways of reasoning about loop behavior.

This led to our derivation of the loop rule, which relies on an *invariant* J.

$$\text{(loop)}\ \frac{\Gamma \vdash J, \Delta \quad J, Q \vdash [\alpha]J \quad J, \neg Q \vdash P}{\Gamma \vdash [\texttt{while}(Q)\,\alpha]P, \Delta}$$

This rule requires us to prove three things in order to show that P holds after the loop executes.

1. $\Gamma \vdash J, \Delta$: The loop invariant J is true when the loop begins its execution.
2. $J, Q \vdash [\alpha]J$: Assuming that both the loop invariant J and the guard Q are true, then executing the body of the loop α one time results in the loop invariant remaining true afterwards.
3. $J, \neg Q \vdash P$: Assuming that both the loop invariant J and the negation of the guard $\neg Q$ are true, the postcondition P must also be true.

Certainly, this is an improvement over reasoning from the axioms, becuase we are able to deduce correctness by considering the behavior of just a single execution of the loop body. However, it leaves open the question of how we should go about finding a suitable loop invariant, and this can be nontrivial. Today we'll take a closer look at this question, using a familiar program as an example.

3 Back to Binary Search

Let's go back to an example that we briefly touched on in the first lecture. Recall that we looked at a buggy version of binary search, shown below.

```
int binarySearch(int key, int[] a, int n) {
   int low = 0;
   int high = n;

   while (low < high) {
       int mid = (low + high) / 2;

       if(a[mid] == key) return mid; // key found
       else if(a[mid] < key) {
           low = mid + 1;
       } else {
            high = mid;
       }
   }
   return -1;  // key not found.
}
```

However, the bug in this case was a bit subtle. If we as the programmer assume that variables assigned type `int` take values from $\mathbb{Z}$, then there is no bug. Actual computers, being nothing more than glorified finite-state machines, of course do not use integers from $\mathbb{Z}$ but rather machine integers of a fixed size. So when the computer executes `low + high` on line 6 in the process of determining the midpoint, the resulting sum may be too large to fit into a machine integer and thus overflow, ultimately causing a negative number to be stored in `mid`.

The simple imperative language that we have studied since does not use machine integers, and we've instead given ourselves the liberty of assuming that all values are integers from $\mathbb{Z}$. This means that if we translate the code from earlier into our language, then we should be able to rigorously prove its correctness using the axioms of Dynamic Logic. However, before we can do so, we will of course need to extend our language to handle arrays.

Basic Arrays: Syntax. Most of us are probably familiar with the square bracket notation for expressing arrays, e.g., `x[i]` to refer to the i^{th} element of the array variable x. Because we are already making heavy use of square brackets in dynamic logic formulas, we will instead opt to use parenthesis for array syntax. We must first add a new term $a(e)$ to the syntax, which corresponds to referencing an array a at the index given by e. Note that for now we allow arbitrary terms to specify the index, so the following are acceptable terms in the new syntax: $a(0), a(1+x), a(a(x)\cdot 3)$.

However, we do need to be careful about distinguishing between variable symbols and array symbols. We don't want to allow certain sets of terms into the language, like $a + a$ and $a \leq a$. To do this, we need to assume that all variable symbols are already defined as either corresponding to arrays or variables.

$$\begin{array}{lll} \text{term syntax} \quad e, \tilde{e} ::= & x & \text{(where } x \text{ is a variable symbol)} \\ & \mid c & \text{(where } c \text{ is a constant literal)} \\ & \mid a(e) & \text{(where } a \text{ is an array symbol)} \\ & \mid e + \tilde{e} & \\ & \mid e \cdot \tilde{e} & \end{array}$$

Note that this definition also prohibits terms like $a(1)(2)$. In a language like C, when looking at this term we might expect that a is an array of arrays, so $a(1)(2)$ obtains the second array stored in a, and then looks up its third value. As we will see when we define the semantics, our language does not have arrays of arrays, so we shouldn't allow terms like this into our language. The above syntax definition accomplishes this.

Recall that our syntax for programs specified assignments as taking the form $x := e$. This means that in order for programs to update arrays, we must also update program syntax with a new alternative for array update. The term $a(e) := \tilde{e}$ does exactly this.

$$\begin{array}{llll}
\text{program syntax} & \alpha, \beta ::= & x := e & \text{(where } x \text{ is a variable symbol)} \\
 & & \mid a(e) := \tilde{e} & \text{(where } a \text{ is an array symbol)} \\
 & & \mid ?Q & \\
 & & \mid \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta & \\
 & & \mid \alpha; \beta & \\
 & & \mid \texttt{while}(Q)\, \alpha &
\end{array}$$

Now that there are arrays in our language, we can write the binary search program in it. We don't have procedures and `return` statements in our language, but we can change things a bit to make it work.

```
$l := 0; h := n;$
$m := (l + h)/2;$
while($l < h \land a(m) \neq k$) {
  if($a(m) < k$)
    $l := m + 1;$
  else
    $h := m;$
  $m := (l + h)/2;$
}
if($l < h$) $r := m$ else $r := -1$
```

Basic Arrays: Term Semantics. Now that we've defined array syntax, we need to give them semantics so that we can reason about their use in programs. Intuitively, we think of arrays as functions from the integers to the type of element stored in the array. We can generalize this idea further by observing that multi-dimensional arrays are nothing more than functions with arity greater than one, from the integers to the element type. This way of modeling generalized array values even extends nicely to constant values, because we can view them as functions of arity zero.

Making this more precise, we will update our semantics for terms and programs to account for all values (i.e., both arrays and integers) as functions of the appropriate arity. We won't worry about multi-dimensional arrays for now (this could return as an exercise), so all of the values we work with will be functions of arity 0 (i.e., integer constants) or 1 (i.e., arrays storing integers). To make this change, we start with redefining the set of all states $\mathcal{S}$. We had previously defined $\mathcal{S}$ as a function that assigns an integer value in $\mathbb{Z}$ to every variable in V, the set of all variables. Now we will define $\mathcal{S}$ to be a function that maps V to the set of functions over $\mathbb{Z}$ of arity at most 1:

$$\mathcal{S} = (\mathbb{Z}^0 \mapsto \mathbb{Z}) \cup (\mathbb{Z}^1 \mapsto \mathbb{Z}) = \mathbb{Z} \cup \mathbb{Z} \mapsto \mathbb{Z}$$

Note that with this change, we can still view the semantics of terms $\omega[\![e]\!]$ as a subset of $\mathcal{S}$, and that of programs $[\![\alpha]\!]$ as a subset of $\mathcal{S} \times \mathcal{S}$.

We can now define the semantics of terms with arrays as we have done previously, by inductively distinguishing the shape of a given term e against a state $\omega \in \mathcal{S}$. However, it might first be a good idea to consider how we would like arrays to be used in the language. For instance, do we want to assign meaning to programs that contain constructs like $a_1 + a_2$? Probably not, so we will need to be careful in how we define the semantics to avoid such cases. Intuitively, we will do so by only assigning semantics to terms involving arrays that evaluate to integer constants. While this will prohibit some cases that we might see as useful, such as $a_1 := a_2$ (i.e., copy an entire array) and $a_1 = a_2$ (i.e., compare all elements of two arrays), we can implement such functionality in other ways.

Definition 1 (Semantics of terms with basic arrays)**.** The semantics of a term e in a state $\omega \in \mathcal{S}$ is its value $\omega[\![e]\!]$, defined inductively as follows.

- $\omega[\![c]\!] = c$ for number literals c
- $\omega[\![x]\!] = \omega(x)$
- $\omega[\![a(e)]\!] = \omega(a)(\omega[\![e]\!])$
- $\omega[\![e + \tilde{e}]\!] = \omega[\![e]\!] + \omega[\![\tilde{e}]\!]$
- $\omega[\![e \cdot \tilde{e}]\!] = \omega[\![e]\!] \cdot \omega[\![\tilde{e}]\!]$

The only change is the addition of the term $a(e)$, which corresponds to array lookup. The semantics defines this by looking up a in ω, evaluating e in ω, and then applying the results.

4 Proving Binary Search

Now that we have semantics for terms that mention arrays, we should be able to reason about the correctness of the binary search code from earlier. But what should the specification be?

Specification. As for a precondition, there is one big assumption that this code must make, namely that the array a is already sorted. There are several ways to specify sortedness of arrays, and they all involve quantifiers. Perhaps the most obvious specification would simply state that for all valid positions in the array $0 \le i < n$, every pair $a(i-1)$ and $a(i)$ are in order:

$$\forall i.0 < i < n \rightarrow a(i-1) \le a(i)$$

This specification is fine, but we should think about how we might want to use it later on. In a sequent calculus proof, this precondition will eventually give us $0 < i < n \rightarrow a(i-1) \le a(i)$ (for some i) in the antecedent. This will let us conclude things directly about adjacent elements in the array, but if we want to reason about elements that are arbitrarily far away, e.g., to prove that $a(0) \leq a(n-1)$, then we will have to do a bit more work as this fact is not immediate from the precondition. We would need to prove a lemma:

$$\vdash (\forall i. 0 < i < n \to a(i-1) \leq a(i)) \to (\forall i_1, i_2. 0 \leq i_1 \leq i_2 < n \to a(i_1) \leq a(i_2))$$

However, because we are free to place whatever we would like in the precondition (within reason), we can simply use the formula on the right as our precondition for sortedness.

$$\texttt{sorted}(a,n) \equiv \forall i_1, i_2. 0 \leq i_1 \leq i_2 < n \to a(i_1) \leq a(i_2)$$

Implicit in this is another precondition, namely that $0 < n$. If we did not have this, then someone could initialize n to be negative, which would cause `sorted` to evaluate to true but result in meaningless program behavior. So, our precondition is:

$$\texttt{pre}(a,n) \equiv 0 < n \land \texttt{sorted}(a,n)$$

Now we need a postcondition. Looking at the program text, the variable r is used to store the result on the last line. If $l < h$, which means that the loop ended early after finding an element with value k, then r takes the position of the element. Otherwise, r takes the value -1. This suggests a postcondition with two cases:

$$\begin{array}{ll} 0 \leq r \to a(r) = k & k \text{ found at position } r \\ r < 0 \to \forall i. 0 \leq i < n \to a(i) \neq k & k \text{ not found} \end{array}$$

The antecedents are mutually exclusive, so we can simply conjoin these cases to arrive at our postcondition:

$$\texttt{post}(a,r,k,n) \equiv (0 \leq r \to a(r) = k) \land (r < 0 \to \forall i. 0 \leq i < n \to a(i) \neq k)$$

The dynamic logic formula that we would then like to prove is:

$$
\begin{array}{l}
\texttt{pre}(a,n) \to [\\
\qquad \left.\begin{array}{l}
\quad \left.\begin{array}{l} l := 0;\\ h := n;\\ m := (l+h)/2; \end{array}\right\}\gamma\\
\texttt{while}(l < h \land a(m) \neq k)\,\{\\
\quad \left.\begin{array}{l} \texttt{if}(a(m) < k)\\ \quad l := m+1;\\ \texttt{else}\\ \quad h := m;\\ m := (l+h)/2; \end{array}\right\}\beta\\
\}\\
\texttt{if}(l < h)\, r := m\, \texttt{else}\, r := -1
\end{array}\right\}\alpha\\
\qquad]\ \texttt{post}(a,r,k,n)
\end{array}
$$

Notice that we will use α as shorthand for the entire program, and β for the portion within the loop, and γ for the first three assignments.

Finding a loop invariant. Before we can begin proving this formula valid, we need to think about a loop invariant. What should it be? It isn't immediately clear from inspection, so perhaps we can find a somewhat systematic way to nudge us in the right direction. One approach that often works is to start writing the proof with a placeholder for the loop invariant, so that we can see what is needed of the loop invariant to make the proof work.

$$\to\text{R}\;\dfrac{[;]\;\dfrac{[\text{if}]\;\dfrac{\texttt{pre}(a,n) \vdash [\gamma][\texttt{while}(l < h \land a(m) \neq k)\,\beta]P}{\texttt{pre}(a,n) \vdash [\gamma][\texttt{while}(l < h \land a(m) \neq k)\,\beta][\texttt{if}(l < h)\,r := m\,\texttt{else}\,r := -1]\texttt{post}(a,r,k,n)}}{\texttt{pre}(a,n) \vdash [\alpha]\texttt{post}(a,r,k,n)}}{\vdash \texttt{pre}(a,n) \to [\alpha]\texttt{post}(a,r,k,n)}$$

In the preceeding, we completed the proof up to the `while` statement. The condition P, which is too large to fit on the page with the surrounding formula, is:

$$P \equiv \underbrace{l < h \to [r := m]\texttt{post}(a,r,k,n)}_{P_T} \land \underbrace{l \geq h \to [r := -1]\texttt{post}(a,r,k,n)}_{P_F}$$

However, we haven't gotten to the loop yet, so we need to continue by applying the loop rule.

$$\text{loop}\;\dfrac{\texttt{pre}(a,n) \vdash [\gamma]J \qquad J, l < h \land a(m) \neq k \vdash [\beta]J \quad J, \neg(l < h \land a(m) \neq k) \vdash P}{\texttt{pre}(a,n) \vdash [\gamma][\texttt{while}(l < h \land a(m) \neq k)\,\beta]P}$$

Of the three things that we must prove, the last corresponding to the invariant implying the postcondition will probably be the most helpful in learning more about a suitable loop invariant. We'll proceed with proving this obligation, and see what we can learn from the attempt.

$$\land\text{R}\;\dfrac{\to\text{R}\;\dfrac{\text{①}}{J, \neg(l < h \land a(m) \neq k) \vdash P_T} \quad \to\text{R}\;\dfrac{\text{②}}{J, \neg(l < h \land a(m) \neq k) \vdash P_F}}{J, \neg(l < h \land a(m) \neq k) \vdash P_T \land P_F}$$

We'll continue with the proof ① of $J, \neg(l < h \land a(m) \neq k) \vdash P_T$ first.

$$[:=]\;\dfrac{\neg\text{L}\;\dfrac{\land\text{R}\;\dfrac{\text{id}\;\dfrac{*}{J, l < h \vdash l < h, \texttt{post}(a,m,k,n)} \quad \neg\text{R}\;\dfrac{\text{③}}{J, l < h \vdash a(m) \neq k, \texttt{post}(a,m,k,n)}}{J, l < h \vdash l < h \land a(m) \neq k, \texttt{post}(a,m,k,n)}}{J, \neg(l < h \land a(m) \neq k), l < h \vdash \texttt{post}(a,m,k,n)}}{J, \neg(l < h \land a(m) \neq k), l < h \vdash [r := m]\texttt{post}(a,r,k,n)}$$

And continuing with ③ from above, let $Q \equiv J, l < h, a(m) = k$:

$$\land\text{R}\;\dfrac{\to\text{R}\;\dfrac{\text{id}\;\dfrac{*}{Q, 0 \leq m \vdash a(m) = k}}{Q \vdash 0 \leq m \to a(m) = k} \quad \to\text{R}\;\dfrac{\neg\text{L}\;\dfrac{\dfrac{\vdots}{Q \vdash 0 \leq m, \forall i. 0 \leq i < n \to a(i) \neq k}}{Q, m < 0 \vdash \forall i. 0 \leq i < n \to a(i) \neq k}}{Q \vdash m < 0 \to \forall i. 0 \leq i < n \to a(i) \neq k}}{Q \vdash (0 \leq m \to a(m) = k) \land (m < 0 \to \forall i. 0 \leq i < n \to a(i) \neq k)}$$

It doesn't seem that we can go any further. From our loop invariant J, in addition to $l < h \land a(m) = k$, we must prove that either $0 \le m$ or that none of the elements of a from 0 to $n - 1$ match k. It shouldn't take too much thought to realize which of these goals will make a more reasonable addition to the loop invariant, because we expect that m should stay within the bounds $[0, n]$ throughout the execution of the loop.

In fact, a bit of thought says that we can strengthen this a bit if we're careful, and stronger invariants make for more direct proofs. In particular, it seems that most of the time, $0 \le m < n$ (note, not just $0 \le m \le n$). Scanning through the code, the only situation that we might get into trouble with this invariant is if $l = h = n$. But we can account for this by observing that as long as $l < h$, then $0 \le m < n$. We'll add this to our invariant.

$$J \equiv l < h \to 0 \le m < n$$

Let's finish the proof of ③, from where we left off at the vertical dots. Let $\Delta \equiv \forall i. 0 \le i < n \to a(i) \ne k$.

$$\to\!\text{L}\;\dfrac{\text{id}\;\dfrac{*}{l < h, a(m) = k \vdash l < h, 0 \le m, \Delta} \qquad \mathbb{Z}\;\dfrac{*}{0 \le m < n, l < h, a(m) = k \vdash 0 \le m, \Delta}}{l < h \to 0 \le m < n, l < h, a(m) = k \vdash 0 \le m, \Delta}$$

At this point, we have shown that the loop invariant $J \equiv l < h \to 0 \le m < n$ implies the postcondition when the program takes the true branch on the final conditional statement (e.g., $l < h$ when the loop terminates). We still need to think about what will happen when it takes the other branch, when $l \ge h$. Let's proceed with the proof, corresponding to the subtree ② from above.

$$[:=]\;\dfrac{\neg\text{L}\;\dfrac{\land\text{R}\;\dfrac{\dfrac{\textcircled{4}}{J, l \ge h \vdash l < h, \mathtt{post}(a, -1, k, n)} \qquad \dfrac{\textcircled{5}}{J, l \ge h \vdash a(m) \ne k, \mathtt{post}(a, -1, k, n)}}{J, l \ge h \vdash l < h \land a(m) \ne k, \mathtt{post}(a, -1, k, n)}}{J, \neg(l < h \land a(m) \ne k), l \ge h \vdash \mathtt{post}(a, -1, k, n)}}{J, \neg(l < h \land a(m) \ne k), l \ge h \vdash [r := -1]\mathtt{post}(a, r, k, n)}$$

In subtree ④, we will need to prove that the postcondition holds for $r = -1$ given J and $l \ge h$, because we cannot prove that $l < h$ given these assumptions. In subtree ⑤, we must prove that either the postcondition holds at $r = -1$, or that $a(m) \ne k$. Let's start with ④.

$$\land\text{R}\;\dfrac{\to\!\text{R}\;\dfrac{\neg\text{L}\;\dfrac{\mathbb{Z}\;\dfrac{*}{J, l \ge h \vdash -1 < 0, l < h, a(-1) = k}}{J, l \ge h, 0 \le -1 \vdash l < h, a(-1) = k}}{J, l \ge h \vdash l < h, 0 \le -1 \to a(-1) = k} \qquad \to\!\text{R}\;\dfrac{\forall\text{R}\;\dfrac{\to\!\text{R}\;\dfrac{\dfrac{\vdots}{J, l \ge h, 0 \le i < n \vdash l < h, a(i) \ne k}}{J, l \ge h \vdash l < h, 0 \le i < n \to a(i) \ne k}}{J, l \ge h \vdash l < h, \forall i. 0 \le i < n \to a(i) \ne k}}{J, l \ge h \vdash l < h, -1 < 0 \to \forall i. 0 \le i < n \to a(i) \ne k}}{J, l \ge h \vdash l < h, (0 \le -1 \to a(-1) = k) \land (-1 < 0 \to \forall i. 0 \le i < n \to a(i) \ne k)}$$

We can't go any further at this point. Our invariant will need to give us enough to prove that when $l \ge h$ and $0 \le i < n$, then $a(i) \ne k$. With this in mind, we might

Aside: Rules for quantifiers

Our proof of ④ used a rule that we have not seen before: $\forall R$. The rule allows us to remove the quantifier, replacing the bound variable with a new variable that does not appear anywhere else in the sequent. This is equivalent to saying that if we can prove that $F(y)$ holds on some y for which we make no prior assumptions, then we can conclude that it holds universally. The corresponding left rule ($\forall L$) says that if we can prove something assuming F holds for a particular term, say e, then we can prove it assuming that F holds universally. Intuitively, we've only made our assumptions stronger by assuming that F holds universally.

$$(\forall\text{L})\ \frac{\Gamma, F(e) \vdash \Delta}{\Gamma, \forall x.F(x) \vdash \Delta} \quad (\forall\text{R})\ \frac{\Gamma \vdash F(y), \Delta}{\Gamma \vdash \forall x.F(x), \Delta} \quad (y \text{ new})$$

The rules for existential quantifiers are similar, but in this case, it is the left rule in which we need to be careful about renaming. Similarly to the $\forall R$, if we can use the fact that $F(y)$ holds to prove Δ, and nothing in our assumptions or Δ mentions specific things about y, then we can conclude that the details of y don't matter for the conclusion, and the only important fact is that some value establishing $F(y)$ exists. The $\exists R$ simply says that if we can prove that F holds for term e, then we can conclude that it must hold for some value, even if we leave the value unspecified.

$$(\exists\text{L})\ \frac{\Gamma, F(y) \vdash \Delta}{\Gamma, \exists x.F(x) \vdash \Delta} \quad (y \text{ new}) \quad (\exists\text{R})\ \frac{\Gamma \vdash F(e), \Delta}{\Gamma \vdash \exists x.F(x), \Delta}$$

go back to our binary search code and think about how the main loop body works. It begins with pointers to the beginning and end of the array, and works its way inwards. At each iteration of the loop, it checks the midpoint of these pointers, and moves one of the pointers to the current midpoint (or its immediate successor) depending on the outcome of the test $a(m) < k$. After each move, we know that because the array is sorted, none of the positions outside the pointers will contain a match for k.

Taking stock of this, our loop invariant might state that for all $0 \le i < n$, whenever $a(i) = k$ then it is the case that $l \le i < h$. We can formalize this, and add it to our invariant.

$$\begin{aligned}
\texttt{mbound} &\equiv l < h \to 0 \le m < n \\
\texttt{notfound} &\equiv \forall i.0 \le i < n \to a(i) = k \to l \le i < h \\
J &\equiv \texttt{mbound} \land \texttt{notfound}
\end{aligned}$$

Now let's see if we can finish off the proof of ④.

$$
{\scriptstyle \forall\mathrm{L}}\,\dfrac{{\scriptstyle \to\mathrm{L}}\,\dfrac{\textcircled{6}\quad {\scriptstyle \to\mathrm{L}}\,\dfrac{\textcircled{7}\quad {\scriptstyle \neg\mathrm{R}}\,\dfrac{{\scriptstyle \mathbb{Z}}\,\dfrac{{\scriptstyle \neg\mathrm{L}}\,\dfrac{{\scriptstyle \mathrm{id}}\,\dfrac{*}{\texttt{mbound}, l \ge h, 0 \le i < n \vdash l \ge h, a(i) \ne k}}{\texttt{mbound}, l < h, l \ge h, 0 \le i < n \vdash a(i) \ne k}}{\texttt{mbound}, l \le i < h, l \ge h, 0 \le i < n \vdash a(i) \ne k}}{\texttt{mbound}, l \le i < h, l \ge h, 0 \le i < n \vdash l < h, a(i) \ne k}}{\texttt{mbound}, a(i) = k \to l \le i < h, l \ge h, 0 \le i < n \vdash l < h, a(i) \ne k}}{\texttt{mbound}, 0 \le i < n \to a(i) = k \to l \le i < h, l \ge h, 0 \le i < n \vdash l < h, a(i) \ne k}}{\texttt{mbound}, \texttt{notfound}, l \ge h, 0 \le i < n \vdash l < h, a(i) \ne k}
$$

The proof of ⑥ is direct from id:

$$
{\scriptstyle \mathrm{id}}\,\dfrac{*}{\texttt{mbound}, l \ge h, 0 \le i < n \vdash 0 \le i < n, l < h, a(i) \ne k}
$$

The proof of ⑦ is nearly as straightforward:

$$
{\scriptstyle \neg\mathrm{R}}\,\dfrac{{\scriptstyle \mathrm{id}}\,\dfrac{*}{\texttt{mbound}, l \ge h, 0 \le i < n, a(i) \ne k \vdash a(i) = k, l < h, a(i) \ne k}}{\texttt{mbound}, l \ge h, 0 \le i < n \vdash a(i) = k, l < h, a(i) \ne k}
$$

So we have now shown that the loop invariant $\texttt{mbound} \land \texttt{notfound}$ implies the postcondition in the case where the loop terminated because $\neg(l < h)$. We still have to show the case where the loop terminated because $a(m) = k$, which corresponds to subtree 5 from above.

$$
{\scriptstyle \wedge\mathrm{R}}\,\dfrac{\textcircled{8}\quad {\scriptstyle \to\mathrm{R}}\,\dfrac{{\scriptstyle \forall\mathrm{R}}\,\dfrac{{\scriptstyle \to\mathrm{R}}\,\dfrac{{\scriptstyle \forall\mathrm{L}}\,\dfrac{{\scriptstyle \to\mathrm{L}}\,\dfrac{\textcircled{9}\quad {\scriptstyle \to\mathrm{L}}\,\dfrac{\textcircled{a}\quad {\scriptstyle \mathbb{Z}}\,\dfrac{{\scriptstyle \neg\mathrm{L}}\,\dfrac{{\scriptstyle \mathrm{id}}\,\dfrac{*}{\texttt{mbound}, l \ge h, 0 \le i < n \vdash l \ge h, a(m) \ne k, a(i) \ne k}}{\texttt{mbound}, l < h, l \ge h, 0 \le i < n \vdash a(m) \ne k, a(i) \ne k}}{\texttt{mbound}, l \le i < h, l \ge h, 0 \le i < n \vdash a(m) \ne k, a(i) \ne k}}{\texttt{mbound}, a(i) = k \to l \le i < h, l \ge h, 0 \le i < n \vdash a(m) \ne k, a(i) \ne k}}{\texttt{mbound}, 0 \le i < n \to a(i) = k \to l \le i < h, l \ge h, 0 \le i < n \vdash a(m) \ne k, a(i) \ne k}}{\texttt{mbound}, \texttt{notfound}, l \ge h, 0 \le i < n \vdash a(m) \ne k, a(i) \ne k}}{J, l \ge h \vdash a(m) \ne k, 0 \le i < n \to a(i) \ne k}}{J, l \ge h \vdash a(m) \ne k, \forall i. 0 \le i < n \to a(i) \ne k}}{J, l \ge h \vdash a(m) \ne k, -1 < 0 \to \forall i. 0 \le i < n \to a(i) \ne k}}{J, l \ge h \vdash a(m) \ne k, (0 \le -1 \to a(-1) = k) \wedge (-1 < 0 \to \forall i. 0 \le i < n \to a(i) \ne k)}
$$

The proof of ⑧ is identical to the corresponding case in our proof of ④ above, so we won't list it here. The proof of ⑨ is identical to that of ⑥ above. Likewise, the proof of ⓐ is idntical to that of ⑦, so we'll omit it here. Finally, to finish off this proof, we used identical reasoning as in the proof of ④. We ended up with conflicting facts about the natural numbers in our assumptions (i.e., $0 \le i < h$ and $l \ge h$), so applying ¬L closed the proof.

We conclude that this loop invariant is strong enough to give us the desired postcondition, but we still need to prove two things:

1. $\mathtt{pre}(a,n) \vdash [\gamma]J$. That the loop invariant holds when we begin executing the loop.
2. $J, l < h \land a(m) \neq k \vdash [\beta]J$. That each iteration of the loop preserves the invariant.

A quick glance at the first three statements should give us some confidence that the (1) will hold, so we'll hold off on the formal proof until the end. However, (2) is a bit more involved, and may even require strengthening the invariant more.

$$\text{[:=]}\;\dfrac{\text{[if]}\;\dfrac{\textcircled{b}\qquad\textcircled{c}}{J, l < h \land a(m) \neq k \vdash [\mathtt{if}(a(m) < k)\, l := m+1 \,\mathtt{else}\, h := m]J(a,l,h,(l+h)/2,n)}}{J, l < h \land a(m) \neq k \vdash [\beta]J(a,l,h,m,n)}$$

We'll attempt to prove each branch now, corresponding to ⓑ and ⓒ. To aid in variable substitutions, we'll use the following placeholders with variables for `mbound` and `notfound`:

$$\mathtt{mbound}(l,h,m,n)$$
$$\mathtt{notfound}(a,n,k,l,h)$$

The beginning of the proof for ⓑ is:

$$\text{[:=]}\;\dfrac{\to\!\text{R}\;\dfrac{\land\text{R}\;\dfrac{\textcircled{d}\qquad\textcircled{e}}{J, l < h \land a(m) \neq k, a(m) < k \vdash \mathtt{mbound}(m+1,h,(l+h)/2,n) \land \mathtt{notfound}(a,n,k,m+1,h)}}{J, l < h \land a(m) \neq k \vdash a(m) < k \to J(a,m+1,h,(l+h)/2,n)}}{J, l < h \land a(m) \neq k \vdash a(m) < k \to [l := m+1]J(a,l,h,(l+h)/2,n)}$$

It shouldn't be surprising that we need to prove each conjunct in our invariant separately. In fact, it is conventional to split loop invariants along conjuncts into a set of loop invariants, as it facilitates readability. We'll start with ⓓ, which says that m remains bounded between 0 and n (non-inclusive) as long as $l < h$.

$$\land\text{L}\;\dfrac{\to\!\text{R}\;\dfrac{\to\!\text{L}\;\dfrac{\textcircled{f}\qquad\dfrac{\vdots}{0 \leq m < n, \mathtt{notfound}, l < h, a(m) \neq k, a(m) < k, m+1 < h \vdash 0 \leq (l+h)/2 < n}}{l < h \to 0 \leq m < n, \mathtt{notfound}, l < h, a(m) \neq k, a(m) < k, m+1 < h \vdash 0 \leq (l+h)/2 < n}}{J, l < h, a(m) \neq k, a(m) < k \vdash m+1 < h \to 0 \leq (l+h)/2 < n}}{J, l < h \land a(m) \neq k, a(m) < k \vdash m+1 < h \to 0 \leq (l+h)/2 < n}$$

Subtree ⓕ follows immediately from the fact that $l < h$ is in our assumptions. We can't complete the proof without the assumption that $h \leq n$, which we have implicitly assumed by adding $l < h \to 0 \leq m < n$ to our invariant. We'll add the fact to our invariant, giving us

$$J \equiv \mathtt{mbound} \land \mathtt{notfound} \land h \leq n$$

Although the addition of this fact won't invalidate our earlier proofs that the invariant implies the postcondition, we will need to prove that it is preserved by the loop, and holds when the loop is first entered. We'll get back to these obligations later, but for now let's return to the proof of ⓑ, picking up where we left off at the vertical dots with this new fact in hand. The rest of the proof follows from arithmetic, because $l < h \le n$, so $(l+h)/2 < n$.

$$\mathbb{Z}\,\dfrac{*}{h \le n, 0 \le m < n, \mathtt{notfound}, l < h, a(m) \ne k, a(m) < k, m+1 < h \vdash 0 \le (l+h)/2 < n}$$

Now we need to prove ⓔ, which is preservation of $\mathtt{notfound}(a, n, k, m{+}1, h)$. To reduce clutter, we'll remove the assumption $a(m) \neq k$, as it implied by $a(m) < k$.

$$\wedge\mathrm{L}\,\dfrac{\forall\mathrm{R}\,\dfrac{\to\!\mathrm{R}\,\dfrac{\to\!\mathrm{R}\,\dfrac{\to\!\mathrm{L}\,\dfrac{ⓖ \quad \wedge\mathrm{R}\,\dfrac{ⓗ \quad ⓘ}{0 \le m < n, \mathtt{notfound}, h \le n, l < h, a(m) < k, 0 \le i < n, a(i) = k \vdash m+1 \le i < h}}{l < h \to 0 \le m < n, \mathtt{notfound}, h \le n, l < h, a(m) < k, 0 \le i < n, a(i) = k \vdash m+1 \le i < h}}{J, l < h, a(m) < k, 0 \le i < n \vdash a(i) = k \to m+1 \le i < h}}{J, l < h, a(m) < k \vdash 0 \le i < n \to a(i) = k \to m+1 \le i < h}}{J, l < h, a(m) < k \vdash \forall i. 0 \le i < n \to a(i) = k \to m+1 \le i < h}}{J, l < h \wedge a(m) \ne k, a(m) < k \vdash \forall i. 0 \le i < n \to a(i) = k \to m+1 \le i < h}$$

Subtree ⓖ follows immediately from the fact that $l < h$ is already in our assumptions. The last step splits $m+1 \le i < h$ into $m+1 \le i \wedge i < h$, and applies ∧R. However, we can't proceed further, even though our intuition suggests that we should be able to. What's going on? We need to prove that $m+1 \le i < h$, and we know that $a(m) < k$ and that $a(i) = k$. We know that this implies that $m+1 \le i$, but why? Because a is sorted! But we cannot use that fact at the moment, because it isn't in our loop invariant. In retrospect, it should have been obvious that we would need to use this fact in our invariant, because we required it in the precondition. Adding it gives us the following:

$$J \equiv \mathtt{mbound} \wedge \mathtt{notfound} \wedge h \le n \wedge \mathtt{sorted}$$

Let's finish the proof of ⓗ, but first we might think about de-cluttering our assumptions to make the reasoning more clear. From our informal discussion in the previous paragraph, we only seem to need the fact that the loop is sorted, the two comparisons involving $a(m)$ and $a(i)$, and facts about the bounds of m. Let's try to finish the proof with just these.

$$\mathbb{Z}\,\dfrac{\forall\mathrm{L}\,\dfrac{\neg\mathrm{L}\,\dfrac{\neg\mathrm{R}\,\dfrac{\to\!\mathrm{L}\,\dfrac{ⓙ \quad =\!\mathrm{L}\,\dfrac{\mathrm{id}\,\dfrac{*}{a(i) \le a(m), 0 \le m < n, 0 \le i < n, a(i) = k, m \ge i \vdash a(m) \ge a(i)}}{a(i) \le a(m), 0 \le m < n, 0 \le i < n, a(i) = k, m \ge i \vdash a(m) \ge k}}{0 \le i \le m < n \to a(i) \le a(m), 0 \le m < n, 0 \le i < n, a(i) = k, m \ge i \vdash a(m) \ge k}}{0 \le i \le m < n \to a(i) \le a(m), 0 \le m < n, 0 \le i < n, a(i) = k \vdash a(m) \ge k, m < i}}{0 \le i \le m < n \to a(i) \le a(m), 0 \le m < n, 0 \le i < n, a(m) < k, a(i) = k \vdash m < i}}{\forall i_1, i_2. 0 \le i_1 \le i_2 < n \to a(i_1) \le a(i_2), 0 \le m < n, 0 \le i < n, a(m) < k, a(i) = k \vdash m < i}}{\forall i_1, i_2. 0 \le i_1 \le i_2 < n \to a(i_1) \le a(i_2), 0 \le m < n, 0 \le i < n, a(m) < k, a(i) = k \vdash m+1 \le i}$$

Subtree Ⓙ follows directly from the assumptions available in our context. We must now prove the other side of the inequality, $i < h$, corresponding to subtree ⓘ.

$$\text{\scriptsize$\forall$L}\dfrac{\text{\scriptsize$\to$L}\dfrac{\textcircled{k} \qquad \text{\scriptsize$\to$L}\dfrac{\textcircled{l} \qquad \text{\scriptsize$\mathbb{Z}$}\dfrac{*}{0 \leq i < n, l \leq i < h, h \leq n, l < h, a(i) = k \vdash i < h}}{0 \leq i < n, a(i) = k \to l \leq i < h, h \leq n, l < h, a(i) = k \vdash i < h}}{0 \leq i < n, 0 \leq i < n \to a(i) = k \to l \leq i < h, h \leq n, l < h, a(i) = k \vdash i < h}}{0 \leq m < n, \forall i.0 \leq i < n \to a(i) = k \to l \leq i < h, h \leq n, l < h, a(m) \neq k \vdash i < h}$$

Subtrees ⓚ and ⓛ follow immediately from the assumptions in our context. We've now closed out subtree ⓓ, which means that we've proved preservation of `mbound`.

We still need to prove preservation of the `mbound` and `notfound` on the other branch of the conditional (subtree ⓒ), in addition to the parts of the loop invariant that we added while proving `mbound`: $h \leq n$ and `sorted`. Let's first look at the preservation proof for `sorted`. The only free variables in this portion of the invariant are a and n, neither of which are updated by the code. Intuitively, this suggests that the proof should be trivial, which we see from the proof.

$$\text{\scriptsize[:=]}\dfrac{\text{\scriptsize id}\dfrac{*}{\texttt{sorted}, \texttt{mbound}, \texttt{notfound}, h \leq n, l < h, a(m) \neq k \vdash \texttt{sorted}}}{\texttt{sorted}, \texttt{mbound}, \texttt{notfound}, h \leq n, l < h, a(m) \neq k \vdash [l := m + 1]\texttt{sorted}}$$

Similarly for $h \leq n$, because h isn't updated on this branch of the conditional, we get preservation by assuming $h \leq n$ as part of the invariant.

Let's continue for now with subtree ⓔ.

$$\text{\scriptsize[:=]}\dfrac{\text{\scriptsize$\wedge$R}\dfrac{\textcircled{m} \quad \textcircled{n} \quad \textcircled{o} \quad \textcircled{p}}{J, l < h, a(m) > k \vdash \texttt{mbound}(l, m, (l + m)/2, n) \wedge \texttt{notfound}(a, n, k, l, m) \wedge m \leq n \wedge \texttt{sorted}}}{J, l < h, a(m) > k \vdash [h := m]J(a, l, m, (l + h)/2, n)}$$

As in the previous branch, because a and n aren't updated, the proof of ⓟ will be similarly trivial. The proof of ⓞ, which is preservation of $h \leq n$, is not as trivial this time. Because h was updated to m, we need to show that:

$$\begin{array}{ll} l < h \to 0 \leq m < n, \forall i.0 \leq i < n \to a(i) = k \to l \leq i < h, & \\ h \leq n, \forall i_1, i_2.0 \leq i_1 \leq i_2 < n \to a(i_1) \leq a(i_2), & \vdash m \leq n \\ l < h, a(m) > k & \end{array}$$

This follows from applying →L on $l < h \to 0 \leq m < n$, and the fact that $l < h$ is one of the premises. However, it should be clear that manually proving all of the obligations is a very laborious undertaking! Once the correct loop invariant has been identified, then the rest of the work is difficult only insofar as it requires a lot of work. This is exactly why we use SMT solvers to prove straightforward formulas like the one we just discussed, and verification condition generators to apply the axioms of dynamic logic to eliminate box and diamond terms.

The remaining obligations ⓜ and ⓝ are left as an exercise, as is the proof that the first three statements in γ establish the invariant, $\texttt{pre}(a, n) \vdash [\gamma]J$.