15-414: Bug Catching: Automated Program Verification

Lecture Notes on Arrays with Updates

Matt Fredrikson André Platzer

Carnegie Mellon University
Lecture 7

1 Introduction

In the previous lecture, we introduced arrays into the language. We modeled arrays as functions, and updated the semantics of terms to account for their behavior. We got plenty of practice using array terms in a proof, as we verified the functionality of the familiar binary search routine. Through a fairly lengthy derivation, we arrived at a suitable loop invariant, and proved its correctness.

This entire discussion was missing something crucial about arrays, namely that they can be updated as the program executes and subsequently read. In this lecture, we will continue defining the semantics of arrays, focusing on updates. We will introduce a notion of function updating that we will use to model array updates, and see that this allows us to avoid the tricky matter of aliasing to some extent. We will then introduce axioms to help us reason about programs that update arrays, and gain some further experience using the axioms on a non-trivial program.

2 Recall: Arrays as Functions

Last lecture we introduced arrays into our language. Syntactically, this was not a huge change, and involved adding a new term for array lookup, as well as a new statement form for array assignment. However, in order to distinguish between variables that store constant values and those that store arrays, we need to assume that all variable symbols are already defined for either arrays or variables. Importantly, this means that there are new sorts of ill-formed terms, for example if x is a variable symbol then $x(1)$ is not a valid term. Similarly, array symbols can only appear in indexed lookup terms,

so if a is an array symbol then a, $a + a$, $a < a$, ... are not valid terms.

term syntax	$e, \tilde{e}$ ::=	x	(where x is a variable symbol)
		$\mid c$	(where c is a constant literal)
		$\mid a(e)$	(where a is an array symbol)
		$\mid e + \tilde{e}$	
		$\mid e \cdot \tilde{e}$	
program syntax	α, β ::=	$x := e$	(where x is a variable symbol)
		$\mid a(e) := \tilde{e}$	(where a is an array symbol)
		$\mid ?Q$	
		$\mid \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta$	
		$\mid \alpha; \beta$	
		$\mid \texttt{while}(Q)\, \alpha$	

Semantically, we modeled arrays as functions from their domain ($\mathbb{Z}$) to their range ($\mathbb{Z}$), which meant that the states of our programs are maps from the set of all variables to $\mathbb{Z} \cup (\mathbb{Z} \to \mathbb{Z})$. We then defined the semantics of terms with arrays in them.

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

Today we'll continue defining the semantics of array updates, and of course see how to reason about programs that use this feature.

3 Array Updates

Basic Arrays: Program Semantics. Now we move to the semantics of programs with arrays. We will need to add a new rule to the inductive definition that accounts for statements of the form $[\![a(e) := \tilde{e}]\!]$, which update an array at a specified position. Operationally, this statement leaves the array a unchanged except at the position given by the valuation of e, which afterwards takes the valuation of $\tilde{e}$.

To specify this behavior, we introduce the *function-patching* notation, which is defined as follows. If A and B are sets, and $f : A \to B$, then for $x, y \in X$ and $b \in B$ we write f_x^b as the function defined by:

$$f_x^b(y) = \begin{cases} b & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$$

Intuitively, f_x^d simply updates f at x to the value d, leaving the rest of f untouched. We can apply function patching multiple times, and the convention is to read the updates from right to left. So for example,

$$f_{x\,y_x}^{1\,0\,2}(x) = 2$$

The rightmost update is the one that is used on application, and any earlier updates are effectively dead.

Definition 2 (Semantics of programs with basic arrays)**.** Each program α is interpreted semantically as a binary reachability relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ over states. The definition is idential to Definition 7 in Lecture 3, except on the following commands:

1. $$[\![a(e) := \tilde{e}]\!] = \{(\omega, \nu) : \omega = \nu \text{ except } \nu[\![a]\!] = \omega(a)^{\omega[\![\tilde{e}]\!]}_{\omega(a)[\omega[\![e]\!]}\}$$

   This is similar to the rule for simple assignment, but in the new state a maps to a function that has been updated at the position given by $\omega[\![e]\!]$ to take the value $\omega[\![\tilde{e}]\!]$. Intuitively, we update the target with an entirely new function, which happens to be the same as $\omega(a)$ everywhere except at $\omega[\![e]\!]$.

Array Axioms. We'd like to reason about programs that make full use of arrays, i.e., programs that update and subsequently read from arrays. The first two axioms, called the read-over-write axioms, have to do with reading from updated arrays.

$$\text{(row1)}\ \ a_e^{\tilde{e}}(e) = \tilde{e}$$

$$\text{(row2)}\ \ e \neq \tilde{e} \to a_e^{e'}(\tilde{e}) = a(\tilde{e})$$

The first read-over-write axiom, row1, establishes that a term that reads from the index that was most recently updated is equal to the value used in the update. The second, row2, establishes that a term that reads from an index that does not match that most recently updated is equal to the value at that index in the array before this update. Let's see how to use them in a proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{a(i) = x, i \neq j \vdash i \neq j, a_i^x(j) = a(j)}\,\text{id}}{a(i) = x \vdash i = j, i \neq j, a_i^x(j) = a(j)}\,\neg\text{R}}{a(i) = x \vdash i = j \vee i \neq j, a_i^x(j) = a(j)}\,\vee\text{R} \quad \dfrac{\text{ⓐ}}{a(i) = x, i = j \vee i \neq j \vdash a_i^x(j) = a(j)}\,\vee\text{L}}{a(i) = x \vdash a_i^x(j) = a(j)}\,\text{cut}}{a(i) = x \vdash \forall j. a_i^x(j) = a(j)}\,\forall\text{R}$$

We continue with subtree @ below.

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{a(i)=x, i=j \vdash x=a(i)}\,\text{id}}{a(i)=x, i=j \vdash a_i^x(i)=a(i)}\,\text{row1}}{a(i)=x, i=j \vdash a_i^x(j)=a(j)}\,\text{=L} \qquad \dfrac{*}{a(i)=x, i\neq j \vdash a_i^x(j)=a(j)}\,\text{row2}}{a(i)=x, i=j \vee i\neq j \vdash a_i^x(j)=a(j)}\,\vee\text{L}$$

In order to apply the axioms, we needed to use the cut rule to introduce the cases $i = j$, $i \neq j$ into our assumptions. This tactic is common in proofs involving array updates, as we need to consider whether the index being accessed has previously been updated.

Moving on to assignment statements, let's begin by reviewing the axiom for simple assignment of constants to variables.

$$([:=])\quad [x := e]p(x) \leftrightarrow p(e)$$

The most natural thing to do is to simply extend this rule to array updates directly. Perhaps something like:

$$([:=]_{()})\quad [a(e) := \tilde{e}]p(a(e)) \leftrightarrow p(\tilde{e})$$

Does this work? Let's try to use it on a simple formula.

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{a(j)\neq 5 \vdash a(j)\neq 5}\,\text{id}}{a(j)\neq 5 \vdash [i:=j]a(j)\neq 5}\,[:=]}{a(j)\neq 5 \vdash [i:=j][a(i):=5]a(j)\neq 5}\,[:=]_{()}}{a(j)\neq 5 \vdash [i:=j; a(i):=5]a(j)\neq 5}\,[;]$$

That doesn't seem right. We started assuming that $a(j) \neq 5$, and then assigned the element indexed by j to take the value 5, and ended up concluding that $a(j)$ still isn't 5! The problem that we've run into is one of aliasing: we can no longer use textual substitution to reason about updates, because the same object might be referenced by syntactically distinct terms. Indeed, when we write $a(i)$ on the right-hand side of an assignment, the assignment's target totally depends on the current state.

In short, we'll need a different axiom for array updates. Let's go back to our semantics, and consider how we modeled arrays. After an update, the new state is the same as the old one, except the variable a maps to an updated version of the old array. This suggests that an axiom more along the lines of:

$$([:=]_{()})\quad [a(e) := \tilde{e}]p(a) \leftrightarrow p(a_{\tilde{e}}^{\tilde{e}})$$

In other words, when we assign $a(i) := e$, then we must show that p holds when all free occurrences of a are replaced by the patched function a_i^e. As a quick sanity check, let's

make sure that this rule doesn't allow us to complete the proof from before.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{a(j) \neq 5 \vdash 5 \neq 5}{a(j) \neq 5 \vdash a_j^5(j) \neq 5}\,\text{row1}}{a(j) \neq 5 \vdash [i := j]a_i^5(j) \neq 5}\,[:=]}{a(j) \neq 5 \vdash [i := j][a(i) := 5]a(j) \neq 5}\,[:=]_0}{a(j) \neq 5 \vdash [i := j; a(i) := 5]a(j) \neq 5}\,[;]$$

This is as we would hope, we can't prove that $5 \neq 5$. So, by modeling array updates with function patching, we have nicely sidestepped the tricky issue of aliasing. We haven't completely avoided it, as we still need to reason about the equality of array indexing terms, which will often require splitting a proof into cases. But accounting for it in this way ensures that we won't be able to continue with a proof unless we've introduced the relevant cases, and our reasoning will remain sound.

Using the axioms to prove reverse copy. Now let's use the axioms to prove a non-trivial formula, which asserts the correctness of a program that makes a reversed copy of an array in a second array. The postcondition that we want this program to satisfy is:

$$\texttt{eqrev}(a, b, n) \equiv \forall i.0 \leq i < n \rightarrow a(i) = b(n - i - 1)$$

The preconditions for such a method should be minimal. Basically, it would be nice to assume that the input array is defined between indices 0 and n, so $0 < n$. In the following proof, we will assume that the array a is defined between 0 and $n - 1$ inclusive, although this will not be explicit in our proof; for now, we can just take this fact for granted.

$$\texttt{pre} \equiv 0 < n$$

Out of convenience, we will also add to our assumptions that our loop variable i is initialized to 0, as this reduces the number of steps in the proof.

$$0 < n, i = 0 \vdash [\texttt{while}(i < n)\,\{b(n - i - 1) := a(i); i := i + 1\}]\texttt{eqrev}(a, b, n)$$

Of course, the first question that we need to answer is what the loop invariant shall be. This is a fairly simple loop, so it shouldn't be too difficult to derive a suitable invariant by inspection. The loop starts from the beginning of a and the end of b, and copies respective elements from a to b. So on the ith iteration, the last i elements of b match the first i elements of b in reverse order. This gives us:

$$J \equiv \forall j.0 \leq j < i \rightarrow a(j) = b(n - j - 1)$$

This bears close resemblance to our postcondition, as it should considering that the program does little else but establish this fact. Continuing with the proof,

$$\text{loop}\dfrac{\text{ⓑ}\ 0 < n, i = 0 \vdash J \qquad \text{ⓒ}\ J, i < n \vdash [\alpha]J \qquad \text{ⓓ}\ J, i \geq n \vdash \texttt{eqrev}(a, b, n)}{0 < n, i = 0 \vdash [\texttt{while}(i < n)\,\{b(n - i - 1) := a(i); i := i + 1\}]\texttt{eqrev}(a, b, n)}$$

In the above, α corresponds to the loop body. Let's get ⓑ out of the way.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{0 < n, i = 0, 0 \leq j \vdash 0 \leq j, a(j) = b(n-j-1)}\,\text{id}}{0 < n, i = 0, 0 \leq j \vdash i \leq j, a(j) = b(n-j-1)}\,=\text{R}}{0 < n, i = 0, 0 \leq j, j < i \vdash a(j) = b(n-j-1)}\,\neg\text{L}}{0 < n, i = 0, 0 \leq j < i \vdash a(j) = b(n-j-1)}\,\wedge\text{L}}{0 < n, i = 0 \vdash 0 \leq j < i \to a(j) = b(n-j-1)}\,\to\text{R}}{0 < n, i = 0 \vdash \forall j.0 \leq j < i \to a(j) = b(n-j-1)}\,\forall\text{R}$$

Now let's check to make sure that the invariant we chose implies the postcondition, by doing ⓓ.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{ⓔ} \qquad \text{ⓕ}}{0 \leq j < i \to a(j) = b(n-j-1), i \geq n, 0 \leq j < n \vdash a(j) = b(n-j-1)}\,\to\text{L}}{0 \leq j < i \to a(j) = b(n-j-1), i \geq n \vdash 0 \leq j < n \to a(j) = b(n-j-1)}\,\to\text{R}}{J, i \geq n \vdash 0 \leq j < n \to a(j) = b(n-j-1)}\,\forall\text{L}}{J, i \geq n \vdash \forall i.0 \leq i < n \to a(i) = b(n-i-1)}\,\forall\text{R}$$

The subtree ⓔ follows from arithmetic, because we have $0 \leq j$ in our assumptions, and $j < n \wedge n \leq i \to j < i$:

$$\dfrac{\dfrac{*}{i \geq n, 0 \leq j < n \vdash 0 \leq j < i, a(j) = b(n-j-1)}\,\mathbb{Z}}{}$$

The subtree ⓕ follows directly from identity, as we have $a(j) = b(n-j-1)$ in our assumptions:

$$\dfrac{\dfrac{*}{a(j) = b(n-j-1), i \geq n, 0 \leq j < n \vdash a(j) = b(n-j-1)}\,\text{id}}{}$$

Great, our invariant gives us the precondition as we expected it would. Now for the final step of proving that the loop body preserves the invariant, which is the proof of ⓒ above.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{ⓖ}}{0 \leq j < i \to a(j) = b(n-j-1), i < n, 0 \leq j \leq i \vdash a(j) = b^{a(j)}_{n-i-1}(n-j-1)}\,\forall\text{L}}{J, i < n, 0 \leq j \leq i \vdash a(j) = b^{a(j)}_{n-i-1}(n-j-1)}\,\mathbb{Z}}{J, i < n, 0 \leq j < i+1 \vdash a(j) = b^{a(j)}_{n-i-1}(n-j-1)}\,\to\text{R}}{J, i < n \vdash 0 \leq j < i+1 \to a(j) = b^{a(j)}_{n-i-1}(n-j-1)}\,\forall\text{R}}{J, i < n \vdash \forall j.0 \leq j < i+1 \to a(j) = b^{a(j)}_{n-i-1}(n-j-1)}\,[:=]_{()}}{J, i < n \vdash [b(n-i-1) := a(i)]\forall j.0 \leq j < i+1 \to a(j) = b(n-j-1)}\,[:=]}{J, i < n \vdash [b(n-i-1) := a(i); i := i+1]\forall j.0 \leq j < i \to a(j) = b(n-j-1)}$$

At this point it might be a good idea to pause and think about our next step. We have that $0 \leq j \leq i$, and an implication contingent on $0 \leq j < i$. We can't prove this antecedent, but we can case split into $0 \leq j < i$ and $j = i$. The former case will follow fairly directly, as we can apply the consequent of the implication to arrive at our goal. The latter case should be even more direct, as the equality $j = i$ will allow us to move forward with the read-over-write axioms. The following corresponds to ⓖ, where in the following we use the following substitutions:

$$\begin{array}{lcl} \Gamma & \equiv & 0 \leq j < i \rightarrow a(j) = b(n-j-1), i < n, 0 \leq j \leq i \\ \Delta & \equiv & a(j) = b^{a(j)}_{n-i-1}(n-j-1) \end{array}$$

We case split by cutting on $0 \leq j < i \vee i = j$. This follows from arithmetic on the fact that $0 \leq j \leq i$. Now for ⓖ:

$$\dfrac{\mathbb{Z}\dfrac{*}{\Gamma \vdash 0 \leq j < i \vee i = j, \Delta} \quad \vee\text{L}\dfrac{\text{ⓗ} \quad \text{ⓘ}}{\Gamma, 0 \leq j < i \vee i = j \vdash \Delta}}{\Gamma \vdash \Delta}\ \text{cut}$$

Now we've split into ⓗ and ⓘ, which correspond to the cases $0 \leq j < i$ and $i = j$, respectively. Let's finish these off, starting with ⓗ where we will omit $0 \leq j \leq i$ from our assumptions because we are adding $0 \leq j < i$, which implies it:

$$\dfrac{\dfrac{\text{ⓙ} \quad \text{cut}\dfrac{\mathbb{Z}\dfrac{*}{j < i \vdash i \neq j}\ \text{row2}\dfrac{\text{id}\dfrac{*}{a(j) = b(n-j-1), j \neq i \vdash a(j) = b(n-j-1)}}{a(j) = b(n-j-1), j \neq i \vdash a(j) = b^{a(j)}_{n-i-1}(n-j-1)}}{a(j) = b(n-j-1), i < n, 0 \leq j < i \vdash a(j) = b^{a(j)}_{n-i-1}(n-j-1)}}{0 \leq j < i \rightarrow a(j) = b(n-j-1), i < n, 0 \leq j < i \vdash \Delta}\ \rightarrow\text{L}$$

Subtree ⓙ follows directly from our case premise, $0 \leq i < j$. Note that our application of row2 took a very brief shortcut. To apply this rule, we needed to show that $n-j-1 \neq n-i-1$. This follows directly from $i \neq j$, so we saved another cut by applying row2 directly from the premise $i \neq j$.

Now we move on to the final subtree ⓘ:

$$\dfrac{\dfrac{\mathbb{Z}\dfrac{*}{i = j \vdash a(j) = a(j)}}{i = j \vdash a(j) = b^{a(j)}_{n-i-1}(n-i-1)}\ \text{row1}}{i = j \vdash a(j) = b^{a(j)}_{n-i-1}(n-j-1)}\ \text{=L}$$