15-414: Bug Catching: Automated Program Verification

Lecture Notes on Parameters & Ghost State

Matt Fredrikson André Platzer

Carnegie Mellon University
Lecture 13

1 Review: procedures

So far, we've defined a programming language with support for arrays, conditionals, loops, and recursive procedures that take no arguments, return no values, and have access to global state.

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)
		$\mid ?Q$	
		$\mid \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$	
		$\mid \alpha; \beta$	
		$\mid \texttt{while}(Q)\,\alpha$	
		$\mid \texttt{m}()$	

The semantics of a procedure call are given by success approximation, where we define the k^{th} syntactic approximation $\alpha^{(k)}$ of a program α to be:

$$\begin{aligned} \alpha^{(0)} &= \texttt{abort} \\ \alpha^{(k+1)} &= \alpha^{\alpha^{(k)}}_{\texttt{m}()} \end{aligned} \tag{1}$$

Here we recall that $\texttt{abort} \equiv ?\mathit{false}$, with semantics $[\![\texttt{abort}]\!] = \emptyset$. Then we obtain the semantics for a procedure call in Definition 1.

Definition 1 (Semantics of procedure calls (with recursion))**.** Let $\alpha^{(k)}$ be the k^{th} approximation of the program α, where $\alpha^{(0)} \equiv \texttt{abort}$ and $\alpha^{(k+1)} = \alpha^{\alpha^{(k)}}_{\texttt{m}()}$. Then if α is the body of procedure $\texttt{m}$, the semantics of $\texttt{m}()$ are as follows:

$$[\![\texttt{m}()]\!] = \{(\omega, \nu) \ : \ (\omega, \nu) \in \textstyle\bigcup_{k \geq 0} [\![\alpha^{(k)}]\!], \text{ where } \alpha \text{ is the body of } \texttt{m}\} \tag{2}$$

When it comes to reasoning about program with recursive calls, we started with the inline rules, which have box and diamond forms.

$$([\text{inl}]) \quad [\texttt{m}()]P \leftrightarrow [\alpha]P \quad (\alpha \text{ is body of } \texttt{m})$$

$$(\langle\text{inl}\rangle) \quad \langle\texttt{m}()\rangle P \leftrightarrow \langle\alpha\rangle P \quad (\alpha \text{ is body of } \texttt{m})$$

By defining contracts for procedures, we can save work by proving that the procedure matches its contract once, and re-using the proof later on whenever the procedure is invoked with [call].

$$([\text{call}]) \quad \frac{\Gamma \vdash A, \Delta \quad A \vdash [\texttt{m}()]B \quad B \vdash P}{\Gamma \vdash [\texttt{m}()]P, \Delta}$$

However, this still does not tell us how to prove a contract for a recursive procedure, where we must reason about an unbounded number of recursive calls. For this, we introduced $\langle$rec$\rangle$.

$$(\langle\text{rec}\rangle) \quad \frac{\Gamma, (\forall \bar{x}. A \wedge \varphi < n) \rightarrow \langle\texttt{m}()\rangle B \vdash \forall \bar{x}. (A \wedge \varphi = n) \rightarrow \langle\alpha\rangle B, \Delta \quad A \vdash \varphi \geq 0}{\Gamma \vdash A \rightarrow \langle\texttt{m}()\rangle B, \Delta} \quad (n \text{ fresh})$$

Much like with our reasoning about loop convergence, $\langle$rec$\rangle$ has us select a variant term φ. We can then assume that the contract holds when $\varphi < n$ when we show that it holds for $\varphi = n$. This form of inductive reasoning allows us to conclude facts about the behavior of recursive procedures.

2 Ghost state

In the previous lecture, we proved that the factorial procedure (shown below) satisfies the contract $A \equiv n \geq 0, B \equiv y = x!$.

```
proc fact() {
  if(x = 0) { y := 1 }
  else { x := x - 1; fact(); x := x + 1; y := y*n; }
}
```

However, this contract is somewhat lacking. Consider the following incorrect implementation of `fact`, which satisfies the same contract.

```
proc fact() {
  x := 0;
  y := 1;
}
```

It should be clear what the shortcoming of the contract is: we did not specify that the value of x remains unchanged by `fact`. But how can we do this? Certainly it would be silly to write the postcondition $B \equiv x = x \land y = x!$, as the left conjunct refers to the same state as the right and adds nothing.

Consider the following program.

```
x := 5;
z := x;
fact();
```

We know that the body of `fact` does not change the variable z. So with this in mind, we could write the contract:

$$\begin{aligned} A &\equiv x \geq 0 \land x = z \\ B &\equiv y = x! \land x = z \end{aligned}$$

All that we are doing here is "remembering" the initial value that x held, before invoking `fact`. But it's very cumbersome and awkward to do this by modifying the contents of the original program we wished to verify. Is this really necessary?

Consider the following proof rule IA.

$$\text{(IA)}\ \frac{\Gamma \vdash [y := e]P, \Delta}{\Gamma \vdash P, \Delta}\quad (y \text{ new})$$

$$\text{(IA)}\ \frac{\Gamma \vdash \langle y := e\rangle P, \Delta}{\Gamma \vdash P, \Delta}\quad (y \text{ new})$$

IA is essentially the assignment axiom, but "in reverse". It introduces a new assignment into the program that was not present before. The fact that this rule is sound follows from the assignment axiom, which allows us to conclude that $P \leftrightarrow [y := e]P$ because y is not mentioned in P.

This rule allows us to introduce a new fresh variable into our proof that remembers a value at a particular point. Because the variable never existed in the program, but will affect the proof, we call it a "ghost variable". When using ghost variables, it is important to make sure that the proof maintains forward momentum. At this point in the semester, it may have become second nature to immediately apply [:=] whenever you see an assignment statement. This would be counterproductive with a ghost variable, as it would leave us right where we began.

$$\cfrac{\cfrac{\Gamma \vdash P, \Delta}{\Gamma \vdash [y := e]P, \Delta}\,[:=]}{\Gamma \vdash P, \Delta}\,\text{IA}$$

In order to move the proof forward after introducing a ghost variable, use the $[:=]_=$ rule that we introduced in previous lectures.

$$([:=]_=)\ \frac{\Gamma, y = e \vdash P(y), \Delta}{\Gamma \vdash [x := e]P(x), \Delta}\quad (y \text{ new})$$

In fact, we can reduce the tedium of repeating these steps by stating a derived rule that combines these steps into one. Below GI does exactly this: introduces a fresh variable y into the context that remembers the value of a term e.

$$\text{(GI)}\ \frac{\Gamma, y = e \vdash P, \Delta}{\Gamma \vdash P, \Delta}\quad (y \text{ new})$$

Example Let's see how to use ghost variables effectively with an example. We'll prove the crucial parts of bubble sort. Recall that bubble sort works by incrementally pushing smaller elements to the beginning of the list, or equivalently, pushing larger elements to the end. We'll implement this functionality in a procedure called `pushRight`.

```
proc PushRight() {
  k := i;
  while(k > 0 ∧ a(k-1) > a(k)) {
    t := a(j-1);
    a(j-1) :=  a(j);
    a(j) := t;
  }
}
```

The contract for `pushRight` states that if its input array a is already sorted between elements 0 and i (another input parameter), then on termination a is sorted between 0 and $i+1$. However, this is not enough. We would also like to specify that a's contents do not change in anything but order. In other words, the final value of a is a permutation of its initial value. To specify this, we will introduce a ghost variable b into the contract.

$$\begin{aligned} A_1 &\equiv a = b \wedge \texttt{sorted}(a, 0, i) \wedge i < n \\ B_1 &\equiv \texttt{perm}(a, b) \wedge \texttt{sorted}(a, 0, i+1) \end{aligned}$$

The main body of bubble sort is then as follows. It takes as input an array a and variable n, which corresponds to the length of a.

```
proc BubbleSort() {
  i := 1;
  while(i < n) {
    PushRight();
    i := i + 1;
  }
}
```

The contract for `BubbleSort` states that the length of a should be positive, and that the value of a on termination is a permutation of its original value, but sorted between 0 and n. We will need to use another ghost variable to keep track of the value of a on entry to `BubbleSort`, but we need to be careful. If we give this ghost variable the same name as that used in our contract for `PushRight`, we will introduce a collision into our

context. We give it a fresh name c.

$$\begin{aligned} A_2 &\equiv a = c \land 0 < n \\ B_2 &\equiv \texttt{perm}(a, c) \land \texttt{sorted}(a, 0, n) \end{aligned}$$

We now want to prove that:

$$\vdash A_2 \to \langle \texttt{BubbleSort}() \rangle B_2$$

Let's begin. `BubbleSort` is not recursive, so we can begin by applying ⟨inl⟩.

$$\langle;\rangle,\langle:=\rangle= \; \to\text{R},\langle\text{inl}\rangle \; \frac{A_2, i = 1 \vdash \langle \texttt{while}(i < n)\,\{\texttt{PushRight}(); i := i + 1\}\rangle B_2}{A_2 \vdash \langle i := 1; \texttt{while}(i < n)\,\{\texttt{PushRight}(); i := i + 1\}\rangle B_2} \over \vdash A_2 \to \langle \texttt{BubbleSort}() \rangle B_2$$

Now we need to prove that the loop converges, and establishes the postcondition B_2. We could do this by applying var, combining the convergence proof with correctness. However, to keep these somewhat distinct concerns separate in our reasoning, we will prove convergence first using the trivial postcondition true, and prove partial correctness of B_2 separately. We will then combine these proofs using inv∧var.

Continuing with the convergence proof, we want to show:

$$A_2, i = 1 \vdash \langle \texttt{while}(i < n)\,\{\texttt{PushRight}(); i := i + 1\}\rangle \top$$

To continue with var, we need a loop variant and invariant. For the variant, let's try $\varphi = n - i$. Because we're not trying to prove anything about correctness except true, we'll use $J \equiv \mathit{true}$ for our invariant. In the following, α denotes the body of the loop.

$$\text{var}\; \frac{\top\text{R}\,\dfrac{*}{A_2, i = 1 \vdash \top} \quad i < n, n - i = m \vdash \langle \alpha \rangle n - i < m \quad \mathbb{Z}\,\dfrac{*}{i < n \vdash n - i \geq 0} \quad \top\text{R}\,\dfrac{*}{i \geq n \vdash \top}}{A_2, i = 1 \vdash \langle \texttt{while}(i < n)\,\{\texttt{PushRight}(); i := i + 1\}\rangle \top}$$

We must prove that executing the loop body α a single time decreases the variant, i.e., yields a state where $n - i < m$.

$$\langle;\rangle,\langle:=\rangle \; \frac{=\text{R}\,\dfrac{\text{G}\,\dfrac{\mathbb{Z}\,\dfrac{*}{\vdash n - (i+1) < n - i}}{i < n, n - i = m \vdash \langle \texttt{PushRight}() \rangle n - (i+1) < n - i}}{i < n, n - i = m \vdash \langle \texttt{PushRight}() \rangle n - (i+1) < m}}{i < n, n - i = m \vdash \langle \texttt{PushRight}(); i := i + 1 \rangle n - i < m}$$

This completes the termination proof. Now we can use the simpler loop rule to prove that the loop establishes B_2. However, in this case we will need a stronger invariant. Let's try $J \equiv \texttt{perm}(a, c) \land \texttt{sorted}(a, 0, i)$.

$$\text{loop}\; \frac{A_2, i = 1 \vdash J \quad J, i < n \vdash [\alpha] J \quad i \geq n, J \vdash B_2}{A_2, i = 1 \vdash [\texttt{while}(i < n)\,\{\texttt{PushRight}(); i := i + 1\}] B_2}$$

The first obligation is to show that the invariant holds when we enter the loop.

$$\wedge\mathrm{R}\,\dfrac{=\mathrm{R}\,\dfrac{\mathbb{Z}\,\dfrac{*}{a=c\vdash \texttt{perm}(a,a)}}{a=c\vdash \texttt{perm}(a,c)}\quad \mathbb{Z}\,\dfrac{*}{a=c,0<n,i=1\vdash \texttt{sorted}(a,0,i)}}{a=c,0<n,i=1\vdash \texttt{perm}(a,c)\wedge\texttt{sorted}(a,0,i)}$$

We know that $\texttt{perm}(a,c)$ holds when $a=c$ because `perm` is reflexive. Similarly, when $i=1$ then $\texttt{sorted}(a,0,i)$ reduces to:

$$\begin{array}{ll} & \forall j,k.0\le j\le k<1\to a(j)\le a(k)\\ \equiv & a(0)\le a(0)\end{array}$$

We move on now to invariant preservation. Because our invariant is conjunctive, we'll first break it into parts.

$$[]\wedge\,\dfrac{J,i<n\vdash[\alpha]\texttt{perm}(a,c)\quad J,i<n\vdash[\alpha]\texttt{sorted}(a,0,i)}{J,i<n\vdash[\alpha](\texttt{perm}(a,c)\wedge\texttt{sorted}(a,0,i))}$$

Proceeding with the preservation of $\texttt{perm}(a,c)$.

$$[;],[:=]\,\dfrac{\mathrm{GI}\,\dfrac{[\texttt{call}]\,\dfrac{J,i<n,a=b\vdash A_1\quad A_1\vdash[\texttt{PushRight}()]B_1\quad B_1\vdash\texttt{perm}(a,c)}{J,i<n,a=b\vdash[\texttt{PushRight}()]\texttt{perm}(a,c)}}{J,i<n\vdash[\texttt{PushRight}()]\texttt{perm}(a,c)}}{J,i<n\vdash[\texttt{PushRight}();i:=i+1]\texttt{perm}(a,c)}$$

Note that we invoked GI before [call]. This is due to the fact that `PushRight`'s precondition stipulates $a=b$. We need to have this in our context to use [call], and the only way to establish it is to introduce the ghost state into the calling context.

Now we need to deal with the contract for `PushRight`. We'll allow ourselves to just assume that $A_1\vdash[\texttt{PushRight}()]B_1$, but we still have obligations for the pre- and post-conditions. However, our invariant and current context gives us everything we need directly, as shown in the following proof.

$$\wedge\mathrm{R}\,\dfrac{\mathrm{id}\,\dfrac{*}{J,i<n,a=b\vdash a=b}\quad \mathrm{id}\,\dfrac{*}{J,i<n,a=b\vdash\texttt{sorted}(a,0,i)}\quad \mathrm{id}\,\dfrac{*}{J,i<n,a=b\vdash i<n}}{J,i<n,a=b\vdash a=b\wedge\texttt{sorted}(a,0,i)\wedge i<n}$$

We now need to show that the postcondition B_1 implies $\texttt{perm}(a,c)$.

$$\texttt{perm}(a,b)\wedge\texttt{sorted}(a,0,i+1)\vdash\texttt{perm}(a,c)$$

Now we have a problem, because we've lost the connection between a and c. We cannot expect the contract for `PushRight` to mention anything about c, as it is nothing more than a ghost variable in this particular calling context. Instead, we see that the current predicament is a result of our lack of foresight prior to invoking [call]. When we introduced b into the calling context and assumed it equal to a, we should have applied =R

to propagate the equality into the postcondition, so that our current obligation would be:

$$\texttt{perm}(a,b) \land \texttt{sorted}(a,0,i+1) \vdash \texttt{perm}(a,b)$$

This is a good tactic to keep in mind when working with ghost variables and procedure contracts. Whenever the contract mentions a ghost variable, there is a good chance that the proof will need to relate that ghost variable to entities in the calling context. We need to return to the proof of $J, i < n \vdash [\texttt{PushRight}(); i := i+1]\texttt{perm}(a,c)$ and set things up to make effective use of the contract A_1, B_1.

However to do this, we will need to utilize a fact about permutations. Namely, that they are transitive. This lemma is provable from the array axioms, although we will omit the proof here for brevity. The lemma gives us:

$$\forall a, b, c. \texttt{perm}(x,y) \land \texttt{perm}(y,z) \to \texttt{perm}(x,z)$$

Then the proof proceeds as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{*\ (\textit{transitivity of}\ \texttt{perm})}{\texttt{perm}(a,b) \land \texttt{perm}(b,c) \vdash \texttt{perm}(a,c)} \quad J, i < n, b = a \vdash [\texttt{PushRight}()](\texttt{perm}(a,b) \land \texttt{perm}(b,c))}{J, i < n, b = a \vdash [\texttt{PushRight}()]\texttt{perm}(a,c)}\ \text{MR}}{J, i < n \vdash [\texttt{PushRight}()]\texttt{perm}(a,c)}\ \text{GI}}{J, i < n \vdash [\texttt{PushRight}(); i := i+1]\texttt{perm}(a,c)}\ [;],[:=]$$

At this point, we seem to have made things even more difficult, because now we need to prove that `PushToRight` gives us both $\texttt{perm}(a,b)$ and $\texttt{perm}(b,c)$. We can use the fact that the box modality distributes over conjunctions to move the goal to:

$$J, i < n, a = b \vdash [\texttt{PushRight}()](\texttt{perm}(a,b) \land \texttt{perm}(b,c))$$

But we still need to show that $\texttt{perm}(b,c)$ holds after the call. We won't have any luck using [call], because the relevant postcondition doesn't mention c. However, we know that $\texttt{perm}(b,c)$ holds *before* the call, and we also know that `PushToRight` doesn't change b or c; these ghost variables are nothing more than a "snapshot" of a at two different points in time, i.e., at the beginning of `BubbleSort` and immediately before the call to `PushToRight`. Instead, we can use the *vacuous axiom*, which says that if a program doesn't modify any of the variables in a formula, and the formula holds before running the program, then it will still hold afterwards.

Theorem 2. *The vacuous axiom is sound:*

$$(V)\ \ P \to [\alpha]P \quad (FV(P) \cap BV(\alpha) = \emptyset)$$

where $FV(P)$ are the free variables in P, and $BV(\alpha)$ are the bound (i.e. written) variables in α. Their intersection must be empty for this axiom to apply.

Continuing on, we now have the following, where β is the body of `PushToRight`.

$$\dfrac{\dfrac{\cdots\ \text{id}\dfrac{*}{B_1 \vdash \texttt{perm}(a,b)}}{[\text{call}]\dfrac{}{J, i < n, a = b \vdash [\texttt{PushRight}()]\texttt{perm}(a,b)}} \quad \text{[inl]}\dfrac{\text{V}\dfrac{*}{J, i < n, a = b \vdash [\beta]\texttt{perm}(b,c)}}{J, i < n, a = b \vdash [\texttt{PushRight}()]\texttt{perm}(b,c)}}{[]\wedge\dfrac{}{J, i < n, a = b \vdash [\texttt{PushRight}()](\texttt{perm}(a,b) \wedge \texttt{perm}(b,c))}}$$

The elided portion of the above proof corresponds to the two obligations that we covered earlier.

Now we move on to the portion of the loop invariant concerned with `sorted`; we must show that it is preserved. This is straightforward, as the postcondition B_1 provides for it explicitly.

$$\dfrac{\dfrac{\dfrac{\dfrac{\textit{see earlier}}{J, i < n, a = b \vdash A_1} \quad \dfrac{\textit{assumed}}{A_1 \vdash [\texttt{PushRight}()]B_1} \quad \text{id}\dfrac{*}{B_1 \vdash \texttt{sorted}(a, 0, i+1)}}{[\text{call}]\dfrac{}{J, i < n, a = b \vdash [\texttt{PushRight}()]\texttt{sorted}(a, 0, i+1)}}}{\text{GI}\dfrac{}{J, i < n \vdash [\texttt{PushRight}()]\texttt{sorted}(a, 0, i+1)}}}{[;],[:=]\dfrac{}{J, i < n \vdash [\texttt{PushRight}(); i := i+1]\texttt{sorted}(a, 0, i)}}$$

To finish the partial correctness proof, we must show that the invariant and negated loop guard imply the postcondition B_2.

$$\wedge\text{L},\wedge\text{R}\dfrac{\text{id}\dfrac{*}{\texttt{perm}(a,c), \texttt{sorted}(a,0,i), i \geq n \vdash \texttt{perm}(a,c)} \quad \mathbb{Z}\dfrac{*}{\texttt{perm}(a,c), \texttt{sorted}(a,0,i), i \geq n \vdash \texttt{sorted}(a,0,n)}}{\texttt{perm}(a,c) \wedge \texttt{sorted}(a,0,i), i \geq n \vdash \texttt{perm}(a,c) \wedge \texttt{sorted}(a,0,n)}$$

Finally, to wrap everything up for a total correctness result, we apply inv∧var. Because we have shown:

$$a = c \wedge 0 < n \rightarrow \langle \texttt{BubbleSort}() \rangle \top$$

in addition to:

$$a = c \wedge 0 < n \rightarrow [\texttt{BubbleSort}()](\texttt{perm}(a,c) \wedge \texttt{sorted}(a,0,n))$$

We can conclude:

$$a = c \wedge 0 < n \rightarrow \langle \texttt{BubbleSort}() \rangle (\texttt{perm}(a,c) \wedge \texttt{sorted}(a,0,n))$$

This completes the proof.