15-414: Bug Catching: Automated Program Verification

Lecture Notes on Parameters & Ghost State

Matt Fredrikson Ruben Martins

Carnegie Mellon University
Lecture 11

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 Procedures with parameters

We'll account for parameters in procedures in the usual way.

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)
		$\mid ?Q$	
		$\mid \texttt{if}(Q)\, \alpha\, \texttt{else}\, \beta$	
		$\mid \alpha; \beta$	
		$\mid \texttt{while}(Q)\, \alpha$	
		$\mid \texttt{m}(e_1, \ldots, e_n)$	

In the call $\texttt{m}(e_1, \ldots, e_n)$, the $e_1, \ldots, e_n$ are called the *actual parameters*. For the corresponding declaration,

```
proc m(x1, ..., xn) {...}
```

the $x_1, \ldots, x_n$ are called the *formal parameters*. The actual parameters are terms that are evaluated in the calling context, using the current state at the moment the call is made. The formal parameters are variables that are assigned the corresponding values of the actuals, for later use in the procedure body. This convention corresponds to the typical call-by-value semantics present in many languages.

When formalizing this, we need to be careful about the state used to evaluate the actuals. We might be tempted to reduce the semantics of a procedure call to a seemingly equivalent sequence of assignments and inlining operations. For example, in the following let α be the body of $\texttt{m}$ and $v_1, \ldots, v_n$ be fresh variables, and assume for the sake of clarity that we are only concerned for the moment with non-recursive procedures.

$$[\![\texttt{m}(e_1, \ldots, e_n)]\!] = [\![v_1 := x_1; \ldots; v_n := x_n; x_1 := e_1; \ldots; x_n := e_n; \alpha; x_1 := v_1; \ldots; x_n := v_n]\!]$$

The idea behind this definition is that before entering the procedure body, the current values of the variables corresponding to the formal parameters are stored in a set of fresh new variables. The formal parameter variables are then assigned to the terms given as actuals, and the procedure body is run. When it completes, the formals are restored to their values before the call.

This definition captures the notion of locality that we want with respect to formal parameters. Namely, that within the procedure they take the values passed in, and the calling context need not worry about variables that happen to collide with formal parameters being overwritten. However, this definition introduces spurious dependencies between the actuals. Consider the following program.

```
proc foo(x, y) {
  ...
}

x := 1;
a := 0;
b := x+2;
foo(a,b);
```

In this example, if we used the above semantics, then the value passed to `foo` in the formal parameter y would be 2, rather than 3 as we would expect.

What we want the semantics to encode is a parallel assignment of all of the formal parameters to their actuals. This leads us to the following definition.

$$\begin{aligned}[\![\texttt{m}(e_1, \ldots, e_n)]\!] = \{(\omega, \nu) \;:\; &\exists\, \mu_1, \mu_2 \text{ where } \mu_1 = \omega \text{ except } \mu_1(x_i) = \omega[\![e_i]\!] \text{ for } i = 1, \ldots, n,\\ &(\mu_1, \mu_2) \in [\![\alpha]\!],\\ &\nu = \mu_2 \text{ except } \nu(x_i) = \omega(x_i) \text{ for } i = 1, \ldots, n\}\end{aligned}$$

This definition properly captures the fact that the formal parameters are local to the procedure body. Let's build intuition for it by using to to evaluate a simple recursive procedure call.

```
proc F(n) {
  if(n == 0)
    y := 0;
  else
    F(n-1);
}
```

Assume that we begin in a state ω_0 and call $F(1)$. The following steps follow from the semantics given above.

1. The state ω_1 shown below is the only state identical to ω_0 except that $\omega_1(n) = \omega_0[\![1]\!] = 1$.
$$\omega_1(v) = \begin{cases} 1 & \text{if } v \text{ is } n \\ \omega_0(v) & \text{otherwise} \end{cases}$$

2. $n = 1$ in ω_1, so we continue by looking for ω_2 where $(\omega_1, \omega_2) \in [\![F(n-1)]\!]$. We see that the only such ω_2 is:
$$\omega_2(v) = \begin{cases} 0 & \text{if } v \text{ is } n \\ \omega_1(v) & \text{otherwise} \end{cases}$$
This is due to the fact that $\omega_1[\![n-1]\!] = \omega_1[\![n]\!] - 1 = 0$.

3. $n = 0$ in ω_2, so we continue by looking for ω_3 where $(\omega_2, \omega_3) \in [\![y := 0]\!]$. This is:
$$\omega_3(v) = \begin{cases} 0 & \text{if } v \text{ is } y \\ \omega_2(v) & \text{otherwise} \end{cases}$$

4. Now we've finished executing the body of the second invocation of F. We look for ω_4 which is identical to ω_3, except it maps the formal argument n to the value that it took in the state that F was invoked in. In this case it was invoked in ω_1, so we have:
$$\omega_4(v) = \begin{cases} \omega_1(n) = 1 & \text{if } v \text{ is } y \\ \omega_3(v) & \text{otherwise} \end{cases}$$
Now we have that $\omega_4(y) = \omega_3(y) = 0$ and $\omega_4(n) = 1$, the value that it took before the actual $n - 1$ was evaluated in ω_1.

5. Finally, we've finished executing the body of the first invocation of F. We finish by finding ω_5 identical to ω_4 except that n is mapped to the value it originally took in ω_0:
$$\omega_5(v) = \begin{cases} \omega_0(n) & \text{if } v \text{ is } y \\ \omega_4(v) & \text{otherwise} \end{cases}$$
So we see that the final state ω_5 is identical to ω_0 except that $\omega_5(y) = 0$, which is what we'd expect from this program.

3 Ghost state

Let's rewrite the factorial procedure with arguments now.

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

In the previous lecture, we proved that the factorial procedure (shown below) satisfies the contract $A \equiv x \geq 0, B \equiv y = x!$. This is a great contract and we should continue to use it, but there is a problem now. Namely, the variable x is local to the procedure, and so it doesn't mean anything in the context from which it is called. To see why this is a problem, suppose that we wanted to prove the validity of:

$$a \geq 0 \land b \geq 0 \rightarrow \langle \texttt{fact}(a+b) \rangle y = (a+b)!$$

Using the $\langle$call$\rangle$ rule, we would begin as follows.

$$\langle\text{call}\rangle \frac{\dfrac{a \geq 0, b \geq 0 \vdash x \geq 0 \qquad x \geq 0 \vdash \langle \texttt{fact}(x) \rangle y = x! \quad y = x! \vdash y = (a+b)!}{a \geq 0, b \geq 0 \vdash \langle \texttt{fact}(a+b) \rangle y = (a+b)!}}{\vdash \langle \texttt{fact}(a+b) \rangle y = (a+b)!} \rightarrow\text{R},\land\text{L}$$

Immediately we are stuck because there is no way to prove that $x \geq 0$ from $a \geq 0$ and $b \geq 0$, much less the postcondition we actually care about. What we need is a way of introducing the fact that when we call the factorial procedure, we set the formal argument x to the term $a+b$, and also a way of remembering that we did this afterwards.

We will see how to do this by the use of "ghost state" in our proof. 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 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.

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

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.

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

Now let's go back to the factorial procedure and see it in action.

$$\text{GI}\ \dfrac{\text{=R}\ \dfrac{\langle\text{call}\rangle\ \dfrac{a \geq 0, b \geq 0, x = a + b \vdash x \geq 0 \qquad x \geq 0 \vdash \langle \texttt{fact}(x)\rangle y = x! \quad y = x! \vdash y = x!}{a \geq 0, b \geq 0, x = a + b \vdash \langle \texttt{fact}(x)\rangle y = x!}}{a \geq 0, b \geq 0, x = a + b \vdash \langle \texttt{fact}(a + b)\rangle y = (a + b)!}}{a \geq 0, b \geq 0 \vdash \langle \texttt{fact}(a + b)\rangle y = (a + b)!}$$

Now the proof looks approachable. The only non-trivial premise is the obligation to prove that the procedure itself satisfies the contract. We need to choose a variant, which we will let $\varphi = x$.

$$\langle\text{rec}\rangle\ \frac{x \geq 0, \forall x.x \geq 0 \land x < n \to \langle \texttt{fact}(x)\rangle y = x! \vdash \forall x.x \geq 0 \land x = n \to \langle \alpha\rangle y = x! \quad x \geq 0 \vdash x \geq 0}{x \geq 0 \vdash \langle \texttt{fact}(x)\rangle y = x!}$$

Let's continue, letting

$$\Gamma \equiv x \geq 0, \forall x.x \geq 0 \land x < n \to \langle \texttt{fact}(x)\rangle y = x!$$

stand for the assumptions from where we left off.

$$\forall\text{R},\to\text{R},\land\text{L}\ \dfrac{\text{[if]}\ \dfrac{\to\text{R},\langle:=\rangle,=\text{R},\mathbb{Z}\ \dfrac{*}{\Gamma, z \geq 0, z = n \vdash z = 0 \to \langle y := 1\rangle y = z!} \quad !\Gamma, z \geq 0, z = n \vdash z > 0 \to \langle \texttt{fact}(z - 1); y := z * y\rangle y = z!}{\Gamma, z \geq 0, z = n \vdash \langle \alpha^z_x\rangle y = z!}}{\Gamma \vdash \forall x.x \geq 0 \land x = n \to \langle \alpha\rangle y = x!}$$

Now the shape of the proof is starting to become clear. We want to use our assumptions to prove that after executing $\texttt{fact}(z - 1)$, $z * y = z!$. To do this, we need to show that

after the recursive call, $y = (z-1)!$. Luckily, our contract provides for exactly this, as reflected in the key assumption from Γ:

$$\forall x. x \geq 0 \land x < n \rightarrow \langle \texttt{fact}(x) \rangle y = x!$$

Because this occurs on the left of the sequent, we will use ∀L to instantiate the quantified variable x as we choose. We instantiate it as $z-1$ to make the proof go through. In the following, let:

$$\Gamma_x^{z-1} \equiv z - 1 \geq 0 \land z - 1 < n \rightarrow \langle \texttt{fact}(z-1) \rangle y = z - 1!$$

Then we continue as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{Z}\ x \geq 0, z > 0, z = n \vdash z - 1 \geq 0 \land z - 1 < n}\,* \qquad \vdots}{x \geq 0, z > 0, z = n, \Gamma_x^{z-1} \vdash \langle \texttt{fact}(z-1) \rangle z * y = z!}\,\rightarrow\text{L}}{\Gamma, z > 0, z = n \vdash \langle \texttt{fact}(z-1) \rangle z * y = z!}\,\forall\text{L}}{\Gamma, z \geq 0, z = n \vdash z > 0 \rightarrow \langle \texttt{fact}(z-1); y := z * y \rangle y = z!}\,\rightarrow\text{R},\langle;\rangle,\langle:=\rangle,\mathbb{Z}$$

Finally, we can close the proof by applying weakening and generalization.

$$\text{WL}\dfrac{\text{M}[\cdot]\dfrac{y = (z-1)! \vdash y * z = z!}{\langle \texttt{fact}(z-1) \rangle y = (z-1)! \vdash \langle \texttt{fact}(z-1) \rangle y * z = z!}}{x \geq 0, z > 0, z = n, \langle \texttt{fact}(z-1) \rangle y = (z-1)! \vdash \langle \texttt{fact}(z-1) \rangle y * z = z!}$$

3.1 Invariants over destructive updates

Another common use of ghost variables is to deal with the fact that some programs change the values of variables that are needed for contracts. An example of this is any sorting procedure, which must modify its input.

```
proc BubbleSort(a, n) {
  i := n-1;
  while(1 ≤ i) {
    PushRight(a, i);
    i := i - 1;
  }
}
```

Any complete specification must say not only that the returned array is sorted, but that it contains the same contents as the original array that was passed in. We capture this below with the `perm` predicate, which is true when its arguments are permutations of eachother. The ghost variable c "memorizes" the original array a so that it can be used in the postcondition.

$$\begin{aligned} A &\equiv \texttt{arrayeq}(a, c) \land 0 < n \\ B &\equiv \texttt{perm}(a, c) \land \texttt{sorted}(a, 0, n) \end{aligned}$$

This is a useful and general tool to use in proofs, and it is not restricted to procedure contracts. Consider the `PushRight` procedure itself.

```
proc PushRight(a, i) {
  j := 0;
  while(j < i) {
    if(a(j) > a(j+1))
      t := a(j+1);
      a(j+1) :=  a(j);
      a(j) := t;
    }
    j := j + 1;
  }
}
```

This procedure increases the range for which a is sorted by one element, and maintains along the way that the array is a permutation of its original contents. A nice way to prove this is to utilize the fact that permutations are transitive.

$$\forall x, y, z.\texttt{perm}(x, y) \wedge \texttt{perm}(y, z) \rightarrow \texttt{perm}(x, z)$$

Namely, we can maintain an invariant which says that a is a permutation of the original (ghosted) c:

$$J \equiv \texttt{perm}(a, c)$$

Then to show that the invariant is preserved, we create a new ghost variable d to hold the contents of a at the beginning of a loop iteration, and prove that the contents of a at the end of the iteration are a permutation of d. The transitive property does the rest.