15-414: Bug Catching: Automated Program Verification

Lecture Notes on Procedures and Ghost State

Matt Fredrikson

Carnegie Mellon University
Lecture 12

1 Introduction

The programs that we have discussed so far are somewhat limited. By restricting the statements allowed in programs to simpler forms, we have been able to understand the fundamental ideas behind the formal semantics and proof techniques for reasoning about program behavior. Importantly, the relative simplicity of the language allowed us to do this without becoming overwhelmed with a significant number of cases and details that need to be considered for rigor, but are not essential to these fundamental ideas.

"Real" programming languages universally support more advanced ways of structuring programs that encourage abstraction, modularity, and reuse. One such construct is the procedure, which gives programmers a way to encapsulate some functionality so that it can be invoked repeatedly in the future. In this lecture we will introduce procedures into the language we have been studying. We will allow our procedures to make use of call-by-value arguments and recursion, but we will not incorporate some of the features widely-used in imperative languages such as local variables and explicit return values.

We will develop ways of reasoning about procedure calls compositionally, using induction to establish contracts for recursive procedures consisting of pre and postconditions along with termination. Having established contracts, we will see how logical monotonicity applies to establish a general rule for making repeated use of previous contracts at call sites. To deal with calls that pass arbitrary expressions as arguments, we will introduce *ghost state* to simplify call expressions by adding additional proof context to keep track of essential relationships across call sites and updates.

2 Review: programs so far

So far, we've defined a fairly simple programming language with support for arrays, conditionals, and loops.

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)**.** The *semantics of a term* e in a state $\omega \in \mathcal{S}$ is its value $\omega[\![e]\!]$. It is defined inductively by distinguishing the shape of term e as follows:

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

Definition 2 (Transition semantics of programs)**.** Each program α is interpreted semantically as a binary reachability relation $[\![\alpha]\!] \subseteq \mathcal{S} \times \mathcal{S}$ over states, defined inductively by

1. $[\![x := e]\!] = \{(\omega, \nu) \,:\, \nu = \omega \text{ except that } \nu[\![x]\!] = \omega[\![e]\!]\}$
   The final state ν is identical to the initial state ω except in its interpretation of the variable x, which is changed to the value that e has in initial state ω.
2. $[\![a(e) := \tilde{e}]\!] = \{(\omega, \nu) : \omega = \nu \text{ except } \nu(a) = \omega(a)\{\omega[\![e]\!] \mapsto \omega[\![\tilde{e}]\!]\}\}$
   The final state ν is identical to the initial state ω except in its interpretation of the array symbol a, which is updated at position $\omega[\![e]\!]$ to take the value $\omega[\![\tilde{e}]\!]$.
3. $[\![?Q]\!] = \{(\omega, \omega) \,:\, \omega \models Q\}$
   The test $?Q$ stays in its state ω if formula Q holds in ω, otherwise there is no transition.

4. $[\![\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta]\!] = \{(\omega,\nu)\ :\ \omega \models Q \text{ and } (\omega,\nu) \in [\![\alpha]\!] \text{ or } \omega \not\models Q \text{ and } (\omega,\nu) \in [\![\beta]\!]\}$
The $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$ program runs α if Q is true in the initial state and otherwise runs β.

5. $[\![\alpha;\beta]\!] = [\![\alpha]\!] \circ [\![\beta]\!] = \{(\omega,\nu)\ :\ (\omega,\mu) \in [\![\alpha]\!], (\mu,\nu) \in [\![\beta]\!]\}$
The relation $[\![\alpha;\beta]\!]$ is the composition $[\![\alpha]\!] \circ [\![\beta]\!]$ of relation $[\![\beta]\!]$ after $[\![\alpha]\!]$ and can, thus, follow any transition of α through any intermediate state μ to a transition of β.

6. $[\![\texttt{while}(Q)\,\alpha]\!] = \big\{(\omega,\nu)\ :$ there are an n and states $\mu_0 = \omega, \mu_1, \mu_2, \ldots, \mu_n = \nu$ such that for all $0 \le i < n$: ① the loop condition is true $\mu_i \models Q$ and ② from state μ_i is state μ_{i+1} reachable by running α so $(\mu_i, \mu_{i+1}) \in [\![\alpha]\!]$ and ③ the loop condition is false $\mu_n \not\models Q$ in the end$\big\}$
The $\texttt{while}(Q)\,\alpha$ loop runs α repeatedly when Q is true and only stops when Q is false. It will not reach any final state in case Q remains true all the time. For example $[\![\texttt{while}(\mathit{true})\,\alpha]\!] = \emptyset$.

3 Adding procedure calls

Now we will extend our language with procedure calls. We'll assume that our language doesn't have any scoping conventions, so procedures can read and modify any variable in the state. To start out, we'll assume that procedures take no arguments, and can modify any variable or array in the state.

We update the program syntax to add a new alternative for procedure call, distinguished by the presence of parenthesis after the procedure name:

program syntax	α, β ::=	$x := e$	(where x is a variable symbol)
		$\mid a(e) := \tilde{e}$	(where a is an array symbol)
		$\mid ?Q$	
		$\mid \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$	
		$\mid \alpha;\beta$	
		$\mid \texttt{while}(Q)\,\alpha$	
		$\mid \texttt{m}(e_1, \ldots, e_n)$	

3.1 First, without recursion

If we can assume that the body of `m` does not make any recursive calls, then we can reason about calls to `m` in a straightforward way.

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. To simplify matters when reasoning compositionally, we will assume that **the only free variables appearing in a procedure are its formal arguments**. This means that the body of a procedure is only allowed to "read" from its arguments, and not from any other variables.

When formalizing the semantics of procedure calls, 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 `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.

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

3.2 Semantics, with recursion

Consider the following procedure that implements the factorial function. It assumes that the variable n contains the "input", and stores the computed factorial value in r.

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

Before going further, let's have another look at the semantics of procedure calls. In non-recursive cases, we can always inline the body of a procedure, replacing all procedure calls with their corresponding bodies, and repeating that process until there are no further opportunities to do so. When we reach this point, we will be left with a program that doesn't have any procedure calls, and we can apply the semantics for other program constructs that we have built up over the semester.

Now that we want to account for recursion, things are different. If we use this process of inlining, we may never finish because each time we replace a call with its body, we introduce at least one more call to the same procedure! Let's formalize this a bit, and see if we can arrive at further insights.

If α is the body of a recursive procedure $\texttt{m}$, then we will use the notation $\alpha^{(k)}$ to denote a *syntactic approximation* of α after k levels of inlining recursive calls. When $k = 0$, we simply replace the entire body with `abort`, which you will recall from earlier is defined as: $\texttt{abort} \equiv ?\mathit{false}$, with semantics $[\![\texttt{abort}]\!] = \emptyset$. To be precise, we define the syntactic approximation $\alpha^{(k)}$ inductively on k as follows:

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

Where $\alpha^{\alpha^{(k)}}_{\texttt{m}()}$ denotes α with all instances of $\texttt{m}()$ replaced with $\alpha^{(k)}$.

We will write $\texttt{m}^{(k)}$ to denote the procedure obtained by replacing the body with $\alpha^{(k)}$. So, for example applying this to the `fact` procedure from before, $\texttt{fact}^{(1)}$ would correspond to the program:

```
proc fact^(1)(n) {
  if(n = 0) { r := 1 }
  else { abort; r := r*n; }
}
```

The second approximation $\texttt{fact}^{(2)}$ would give us:

```
proc fact^(2)(n) {
  if(n = 0) { r := 1 }
  else {
    if(n-1 = 0) { r := 1 }
    else { abort; r := r*n; };
    r := r*n
  }
}
```

In general, we would write $\texttt{fact}^{(k)}$ when $k > 0$ as:

```
proc fact^(k)(n) {
  if(n = 0) { r := 1 }
  else { fact^(k-1)(n-1); r := r*n; }
}
```

Let's think about this in the context of concrete executions of `fact`. If we call `fact`(n) with $n \leq 0$, then we can reason about the behavior of this by considering `fact`$^{(1)}$. The reason is that we will never encounter the "else" branch, and return the correct answer $y = 1 = 0!$. So we know that:

$$[\![\texttt{fact}^{(1)}(n)]\!] \subseteq [\![\texttt{fact}(n)]\!]$$

Similarly, if we call `fact`(n) with $n \leq 1$, then we can reason about the behavior by considering `fact`$^{(1)}$ and `fact`$^{(2)}$. Looking back at the second approximation of `fact` listed above, we see that n will be decremented immediately in the "else" branch, and tested against 0 before the `abort` is ever reached. So we know that:

$$[\![\texttt{fact}^{(1)}(n)]\!] \cup [\![\texttt{fact}^{(2)}(n)]\!] \subseteq [\![\texttt{fact}(n)]\!]$$

Continuing with this line of reasoning, the pattern is clear. If we want to account for the behavior of calling `fact`(n) where n is at most k, then we need to ensure that,

$$[\![\texttt{fact}^{(1)}(n)]\!] \cup \cdots \cup [\![\texttt{fact}^{(k+1)}(n)]\!] \subseteq [\![\texttt{fact}(n)]\!]$$

But we want to characterize the full semantics of `fact`, imposing no upper bound on the value of n from which we call it. We can take our reasoning to its logical conclusion, and define the semantics of a procedure call as shown in Definition 3.

Definition 3 (Semantics of procedure calls (with recursion))**.** The semantics of $\texttt{m}(e_1, \ldots, e_n)$ are as follows:

$$[\![\texttt{m}(e_1, \ldots, e_n)]\!] = \{(\omega, \nu) \ : \ (\omega, \nu) \in \textstyle\bigcup_{k \geq 0} [\![\texttt{m}^{(k)}(e_1, \ldots e_n)]\!]\} \tag{2}$$

Although we did not talk about the inclusion of $\texttt{m}^{(0)}$ on our way to this definition, note that because $[\![\texttt{m}^{(0)}]\!] = \emptyset$, it follows trivially.

4 Reasoning compositionally about procedure calls

Now that we understand the meaning of procedure calls, we will derive reasoning principles for dealing with them in correctness proofs. We will start with a rule that lets of make use of the familiar notion of contracts, and then see how to go about showing that contracts hold for a given procedure.

4.1 Contracts

However, we often know more about procedures because we write contracts, or precondition-postcondition pairs that specify requirements at the call site and guarantees about the state afterwards. If we assume a precondition A and postcondition B for $\mathtt{m}$, then we can avoid having to prove things directly about the body α and instead just show that the contract gives us what we need.

Theorem 4. *The contract procedure call rule is sound by derivation.*

$$([\mathit{call}])\ \frac{\Gamma \vdash A \quad A \vdash [\mathtt{m}(x_1,\ldots,x_n)]B \quad B \vdash P}{\Gamma \vdash [\mathtt{m}(x_1,\ldots,x_n)]P, \Delta}$$

Proof. We can derive this rule using [inl], M[·], and two applications of cut. The principle follows from monotonicity, and is not specific to the program in the box being a procedure call. The proof is as follows.

$$\dfrac{\dfrac{\Gamma \vdash A, \Delta}{\Gamma \vdash [\mathtt{m}(x_1,\ldots,x_n)]P, \Delta}\,\text{cut,WL,WR} \qquad \dfrac{A \vdash [\mathtt{m}(x_1,\ldots,x_n)]B \qquad \dfrac{B \vdash P}{[\mathtt{m}(x_1,\ldots,x_n)]B \vdash [\mathtt{m}(x_1,\ldots,x_n)]P}\,\text{M[·]}}{A \vdash [\mathtt{m}(x_1,\ldots,x_n)]P}\,\text{cut,WL}}{\Gamma \vdash [\mathtt{m}(x_1,\ldots,x_n)]P, \Delta}\,\text{cut,WL,WR}$$

□

Note that because the proof of [call] follows from monotonicity, which also holds for diamond modalities, the corresponding total correctness rule is sound as well.

Theorem 5. *The contract procedure call rule ⟨call⟩ is sound by derivation.*

$$(\langle \mathit{call} \rangle)\ \frac{\Gamma \vdash A \quad A \vdash \langle \mathtt{m}(x_1,\ldots,x_n) \rangle B \quad B \vdash P}{\Gamma \vdash \langle \mathtt{m}(x_1,\ldots,x_n) \rangle P, \Delta}$$

The rules [call] and ⟨call⟩ are convenient in practice because we can decide on the contract A, B once and for all before using the procedure, construct a proof of $A \vdash [\alpha]B$, and reuse that proof whenever we need to reason about a call to $\mathtt{m}$. All that we need to do for each call is derive a proof that the calling context entails the precondition ($\Gamma \vdash A, \Delta$), and a corresponding proof that the postcondition gives the property we're after ($B \vdash P$). This sort of compositionality lets us reuse past work, and is key to scaling verification to larger and more complex programs.

4.2 Total correctness

Now that we've seen how to reason compositionally about procedure calls, let's see how we can go about establishing a contract to begin with. We'll do so in the general case that includes recursion, as non-recursive procedures can be dealt with through sufficient inlining as described above.

We should expect that in order to reason about recursive procedure calls, we will need to use induction. Just as when reasoning about loops we need to find an inductive loop invariant that allows us to conclude things about executions that may continue for arbitrarily many steps, when reasoning about procedures we typically need to find a contract that allows us to reason about the effects of calls, recursive or otherwise.

Recall from our discussion of loop convergence the var rule.

$$\text{(var)}\ \frac{\Gamma \vdash J, \Delta \quad J, Q, \varphi = n \vdash \langle\alpha\rangle(J \wedge \varphi < n) \quad J, Q \vdash \varphi \geq 0 \quad J, \neg Q \vdash P}{\Gamma \vdash \langle \texttt{while}(Q)\, \alpha\rangle P, \Delta}\ (n \text{ fresh})$$

This rule requires that we select a term φ whose value will decrease with each iteration of the loop. It should stay nonnegative as long as the loop continues executing, assuming that we declare 0 to be the lower bound towards which φ converges. As long as it never violates this bound, the invariance of φ's decreasing lets us conclude that the loop terminates.

How can we use similar reasoning with a recursive procedure `m`? Intuitively, we want to associate a similar term φ with each call to `m`, and argue that this term decreases each time `m` makes a recursive call. The rule $\langle$rec$\rangle$ below captures this reasoning, where α is the body of `m` and $\bar{x}$ is the list of `m`'s formal arguments.

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

Intuitively, this rule says that if we want to conclude that `m` terminates in a state described by B when starting in one described by A, then we reason about the body assuming that the variant term $\varphi = n$ when it begins executing. We are allowed to assume that recursive calls beginning in a state where $\varphi < n$ will terminate in one described by B. Importantly, we must also show that the variant φ is bounded below by 0.

Theorem 6 ([AdBO09]). *The rule $\langle$rec$\rangle$ is sound. That is, if we have*

$$\models (\forall \bar{x}. A \wedge \varphi < n \to \langle \texttt{m}(\bar{x})\rangle B) \to (A \wedge \varphi = n \to \langle\alpha\rangle B) \tag{3}$$

and

$$\models A \to \varphi \geq 0 \tag{4}$$

then it is the case that

$$\models A \to \langle \texttt{m}(\bar{x})\rangle B \tag{5}$$

Back to `fact`. Let's apply this to `fact` from before, and prove that it terminates. One minor annoyance is that $\langle$rec$\rangle$ relies on a fresh variable n, which we also used in `fact`. We'll modify the program slightly to ensure that the steps of our proof line up with the symbols in $\langle$rec$\rangle$.

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

Recall the contract we stated earlier.

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

In the proof, we'll use the following shorthand to keep the proof steps more concise:

$$P \equiv \forall x. x \geq 0 \land \varphi < n \to \langle \texttt{fact}(x) \rangle r = x!$$

We have only to decide what to use as a variant. Looking at the code, each time `fact` is called, x decreases from the value that it had on entry to the procedure. So we will use $\varphi \equiv x$. The proof is then as follows, where α denotes the body of `fact`.

$$\dfrac{\dfrac{\dfrac{\text{ⓐ} \quad \langle;\rangle,\langle:=\rangle \dfrac{P, x = n, x > 0 \vdash \langle \texttt{fact}(x-1) \rangle r * x = x!}{P, x = n, x > 0 \vdash \langle \texttt{fact}(x-1); r := r * x \rangle r = x!}}{P, x \geq 0, x = n \vdash \langle \alpha \rangle r = x!} \langle\text{if}\rangle,{\to}\text{R} \quad \text{id} \dfrac{*}{x \geq 0 \vdash x \geq 0}}{\vdash x \geq 0 \to \langle \texttt{fact}(x) \rangle r = x!} \langle\text{rec}\rangle,{\to}\text{R},{\land}\text{L}$$

The proof ⓐ corresponds to the branch of the conditional where $x = 0$. Note that this is the base case of the recursive procedure, and follows easily as seen below:

$$\langle:=\rangle \dfrac{\mathbb{Z} \dfrac{*}{P, x = n, x = 0 \vdash 1 = x!}}{P, x = n, x = 0 \vdash \langle r := 1 \rangle r = x!}$$

Now we continue with the other branch where the main proof left off, where we are ready to deal with the call to `fact`. Here is where the fact that we have universally quantified over x will help, as we can instantiate x with $x - 1$ to reflect the fact that this is the value of the actual argument that the recursive call is given. Note that we leave out the antecedent premise for the application of $\to$L; it follows directly from the assumptions $x = n, x > 0$.

$$\begin{array}{rl} \mathbb{Z} & \dfrac{*}{r = (x-1)! \vdash r * x = x!} \\ \text{M} & \dfrac{}{\langle \texttt{fact}(x-1) \rangle r = (x-1)!, x = n, x > 0 \vdash \langle \texttt{fact}(x-1) \rangle r * x = x!} \\ {\to}\text{L} & \dfrac{}{x - 1 \geq 0 \land x - 1 < n \to \langle \texttt{fact}(x-1) \rangle r = (x-1)!, x = n, x > 0 \vdash \langle \texttt{fact}(x-1) \rangle r * x = x!} \\ \forall\text{L} & \dfrac{}{\forall x. x \geq 0 \land x < n \to \langle \texttt{fact}(x) \rangle r = x!, x = n, x > 0 \vdash \langle \texttt{fact}(x-1) \rangle r * x = x!} \end{array}$$

This completes the proof.

5 Ghost state

One thing to notice about the ⟨call⟩ and ⟨rec⟩ rules is that they only apply to calls done on actuals corresponding to variables, not arbitrary terms. So, for example, if we wanted to prove the following line, we cannot use the ⟨rec⟩ rule, because $a+b$ is not a variable.

$$a \geq 0, b \geq 0 \vdash \langle \texttt{fact}(a+b)\rangle r = (a+b)!$$

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.

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

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.

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

Now the proof is approachable using ⟨call⟩.

$$\dfrac{\dfrac{}{a \geq 0, b \geq 0, x = a + b \vdash x \geq 0}\,\mathbb{Z}^{*} \qquad x \geq 0 \vdash \langle \texttt{fact}(x) \rangle r = x! \qquad \dfrac{}{r = x! \vdash r = x!}\,\text{id}^{*}}{a \geq 0, b \geq 0, x = a + b \vdash \langle \texttt{fact}(x) \rangle r = x!}\,\langle\text{call}\rangle$$

Of course, the middle premise is exactly what we proved earlier when demonstrating the use of ⟨rec⟩. Notice, however, that if we hadn't applied =R after GI earlier, to the $(a+b)$ in the postcondition, then we would not have been able to complete the rightmost premise. The reason is that the context we gained from GI relating x to $a+b$ is no longer available when reasoning about the post-state, due to the fact that ⟨call⟩ is a special case of monotonicity. Oftentimes when using ghost state in a proof, a bit of planning and foresight is needed to set things up to ensure that the proof will eventually close out.

5.1 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.

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

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.

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

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.

6 Summary of today's rules

$$([\text{call}]) \quad \frac{\Gamma \vdash A, \Delta \quad A \vdash [\mathtt{m}(x_1, \ldots, x_n)]B \quad B \vdash P}{\Gamma \vdash [\mathtt{m}(x_1, \ldots, x_n)]P, \Delta}$$

$$(\langle\text{call}\rangle) \quad \frac{\Gamma \vdash A \quad A \vdash \langle\mathtt{m}(x_1, \ldots, x_n)\rangle B \quad B \vdash P}{\Gamma \vdash \langle\mathtt{m}(x_1, \ldots, x_n)\rangle P, \Delta}$$

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

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

References

[AdBO09] Krzysztof R. Apt, Frank de Boer, and Ernst-Rdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 3rd edition, 2009.