15-414: Bug Catching: Automated Program Verification

Lecture Notes on Recursive Procedures

Matt Fredrikson Ruben Martins

Carnegie Mellon University

Lecture 10

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 start with the simplest case: non-recursive procedures that take no explicit arguments, do not provide an explicit return value, and have access to the same variables as the context in which they are called. Even with these restrictions, we will see how to reason about procedure calls compositionally, using contracts, so that we can avoid redundant work in proving things about their behavior.

We will then consider recursive procedures, still with no arguments or return value. The main challenge with recursive procedures lies in proving that their contracts hold, and we will see how to use inductive principles to accomplish this. Finally, we will discuss termination, and learn how to use variant terms to prove this similar to how we were able to reason about loop convergence. In future lectures, we will add arguments and return values to our procedures, and discuss some additional techniques that simplify reasoning about the input/output behavior of such procedures.

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	$\alpha, \beta ::=$	$x := e$	(where x is a variable symbol)
		$\mid a(e) := \tilde{e}$	(where a is an array symbol)
		$\mid ?Q$	
		$\mid \mathtt{if}(Q)\,\alpha\,\mathtt{else}\,\beta$	
		$\mid \alpha; \beta$	
		$\mid \mathtt{while}(Q)\,\alpha$	

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. $[\![\mathtt{if}(Q)\,\alpha\,\mathtt{else}\,\beta]\!] = \{(\omega,\nu) : \omega \models Q \text{ and } (\omega,\nu) \in [\![\alpha]\!] \text{ or } \omega \not\models Q \text{ and } (\omega,\nu) \in [\![\beta]\!]\}$
   The $\mathtt{if}(Q)\,\alpha\,\mathtt{else}\,\beta$ 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. $[\![\mathtt{while}(Q)\,\alpha]\!] = \big\{(\omega,\nu) :$ 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 $\mathtt{while}(Q)\,\alpha$ 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 $[\![\mathtt{while}(\mathit{true})\,\alpha]\!] = \emptyset$.

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:

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

First, no 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. What is the semantics of m()? We can think of simply inlining the body α into the call site.

$$[\![\mathtt{m}()]\!] = \{(\omega,\nu) : (\omega,\nu) \in [\![\alpha]\!], \text{ where } \alpha \text{ is the body of } \mathtt{m}\} \tag{1}$$

Now that we have semantics to work from, we can define an axiom to help us reason about calls. Just as Equation 1 replaces the call with its corresponding body, the axiom substitutes the call for its body.

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

It is easy to show that this axiom is sound using a semantic argument. This axiom can be useful if we know nothing about the behavior of `m`, because we can reason about it as though it weren't invoked by a call in the first place, but instead the program had been written in long form with the body repeated wherever calls appear.

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 `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 3. *The contract procedure call rule is sound by derivation.*

$$(\textit{[call]})\quad \dfrac{\Gamma \vdash A \qquad A \vdash [\mathtt{m}()]B \qquad B \vdash P}{\Gamma \vdash [\mathtt{m}()]P, \Delta}$$

Proof. We can derive this rule using [inl], M[·], and two applications of cut. The proof is as follows.

$$\text{[inl]}\,\dfrac{\text{cut,WL,WR}\,\dfrac{\Gamma \vdash A, \Delta \qquad \text{cut,WL}\,\dfrac{A \vdash [\alpha]B \qquad \mathrm{M}[\cdot]\,\dfrac{B \vdash P}{[\alpha]B \vdash [\alpha]P}}{A \vdash [\alpha]P}}{\Gamma \vdash [\alpha]P, \Delta}}{\Gamma \vdash [\mathtt{m}()]P, \Delta}$$

□

The rule [call] is 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 `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 Dealing with recursion

Now that we've seen how to reason compositionally about non-recursive procedure calls in our code, let's add recursion into the mix. 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 y. Because we don't support arguments yet, we have to manage modifications made by subsequent calls somewhat carefully, which is why `fact` increments n after the recursive call.

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

4.1 Semantics of recursive procedures

Before going further, let's have another look at the semantics of procedure calls. When we stated Equation 1, we were only thinking about non-recursive procedures. In these cases, inlining makes perfect sense: we can imagine 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 `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:

$$\begin{aligned} \alpha^{(0)} &= \texttt{abort} \\ \alpha^{(k+1)} &= \alpha^{\alpha^{(k)}}_{\texttt{m}()} \end{aligned} \tag{2}$$

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

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

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

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

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

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

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

Similarly, if we call `fact()` with $n = 1$, then we can reason about the behavior by considering $\texttt{fact}^{(2)}$. 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:

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

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

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

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 4.

Definition 4 (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 `m`, the semantics of `m()` are as follows:

$$[\![\texttt{m}()]\!] = \{(\omega,\nu) \ : \ (\omega,\nu) \in \textstyle\bigcup_{k\geq 0} [\![\alpha^{(k)}]\!], \text{ where } \alpha \text{ is the body of } \texttt{m}\} \tag{3}$$

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

4.2 Correctness

The obvious contract for `fact` is,

$$A \equiv n \geq 0, B \equiv y = n!$$

We can see right away that applying [call] does not get us very far, because α makes a call to `fact`. We would again need to apply [call], and so on.

To make progress on this, we should recall how we tend to think about the correctness of recursive procedures informally. When writing recursive code, the first thing that we do is determine what the base cases are, for which we can return a value without making further recursive calls. After convincing ourselves that we've accounted for the base cases correctly, we move on to those that do require a recursive call. Each time we need to make such a call, we *assume* that the value we get back will be correct, and use it to ultimately return a new value for the argument we're dealing with.

Our assumption that the procedure returns correct values on smaller arguments, and leveraging that to construct a correct value for larger arguments, is clearly a form of inductive reasoning. What if we construct a rule that lets us assume that a procedure we're considering is correct, so that we can reason that each recursive call is correct as a result? The rule might look something like the following.

$$(\text{[rec]})\ \frac{\Gamma, \forall x_1,\ldots,x_n. A \to [\mathtt{m}()]B \vdash A \to [\alpha]B, \Delta}{\Gamma \vdash A \to [\mathtt{m}()]B, \Delta}\quad (\alpha \text{ is body of } \mathtt{m}, x_1,\ldots,x_n \text{ used in } \alpha)$$

The [rec] rule makes reference to all variables $x_1,\ldots,x_n$ that are used in α. What does this mean, and why is it necessary? First, this requirement means that in the premise, we must quantify over all of the variables that appear in the procedure body α. Because our procedures don't take any arguments, and have access to the same set of variables as the context in which they are invoked, it is appropriate to think of the set of variables that are read in the body as its arguments. Universally quantifying over them in the premise formalizes the notion that no matter what value we give to the arguments to `m` when we call it, as long as the precondition holds (on those values) then the postcondition will as well.

But before we commit to this rule, we must ask ourselves whether it is sound. A search through the verification literature may serve to give us some assurance, as many conference papers, journal articles, and textbooks have introduced this rule as sound. However, consider the following example.

$$\cfrac{\cfrac{\cfrac{\cfrac{*}{n \geq 0 \to [\alpha]y = 42 \vdash n \geq 0 \to [\alpha]y = 42}\,\text{id}}{n \geq 0 \to [\mathtt{fact}()]y = 42 \vdash n \geq 0 \to [\alpha]y = 42}\,\text{[inl]}}{\forall n, y. n \geq 0 \to [\mathtt{fact}()]y = 42 \vdash n \geq 0 \to [\alpha]y = 42}\,\forall\text{L}}{\vdash n \geq 0 \to [\mathtt{fact}()]y = 42}\,\text{[rec]}$$

Using [rec] we showed that all terminating runs of $\mathtt{fact}()$ leave y with the value 42. In fact, notice that we could have substituted any procedure in place of `fact`, and any contract, and written essentially the same proof. So, this rule is certainly not sound. We will need to come up with something else.

5 Total correctness of procedure calls and contracts

So far, we have developed an intuition that in order to reason about recursive procedure calls, we will need to use induction. We have also observed a number of similarities between the reasoning needed for recursive procedures and that needed for loops. 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 needed to find a contract that allowed us to reason about calls, recursive or otherwise.

Recall from our discussion of loop convergence the var rule.

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

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. As with [rec], we will embody this obligation in an assumption about recursive calls that allows us to make conclusions about the procedure body. The rule ⟨rec⟩ below captures this reasoning.

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

Intuitively, this rule says that is 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.

Theorem 5. *The rule ⟨rec⟩ is sound. That is, if we have*

$$\models (\forall\bar{x}.A \land \varphi < n \to \langle\alpha\rangle B) \to (\forall\bar{x}.A \land \varphi = n \to \langle\alpha\rangle B) \tag{4}$$

and

$$\models A \to \varphi \geq 0 \tag{5}$$

then it is the case that

$$\models A \to \langle \texttt{m}()\rangle B \tag{6}$$

Proof. The following proof is adapted from [AdBO09].

Note that because n does not appear in A or φ, we can conclude that $A \equiv \exists n.(A \land \varphi < n)$. So using the fact that $\models A \to \varphi \geq 0$, we have $\exists n.(A \land \varphi < n) \to \exists n \geq 0.(A \land \varphi < n)$. From this we see that it is sufficient to show that the following is implied by (4) and (5).

$$\models \exists n \geq 0.A \land \varphi < n \to \langle \texttt{m}()\rangle B$$

Furthermore, because n does not appear in α or B, $\exists n \geq 0.(A \land \varphi < n) \rightarrow \langle \mathtt{m}() \rangle B$ is true if $A \land \varphi < n \rightarrow \langle \mathtt{m}() \rangle B$ is true for all $n \geq 0$. We then write our goal as follows:

$$\models A \land \varphi < n \rightarrow \langle \mathtt{m}() \rangle B$$

and proceed by induction on n.

Basis $n = 0$**.** We want to show:

$$\models A \land \varphi < 0 \rightarrow \langle \mathtt{m}() \rangle B$$

Fix an arbitrary state ω. We have that $\omega \models A \rightarrow \varphi \geq 0$, so $\omega \not\models A \land \varphi < 0$. This gives us $\omega \models A \land \varphi < 0 \rightarrow \langle \mathtt{m}() \rangle B$.

Induction step $n > 0$**.** We want to show:

$$\models A \land \varphi < n + 1 \rightarrow \langle \mathtt{m}() \rangle B$$

Fix an arbitrary state ω. The inductive hypothesis gives us that $\omega \models A \land \varphi < n \rightarrow \langle \alpha \rangle B$. From this, (4) and $\langle \text{inl} \rangle$, we have that $\omega \models A \land \varphi = n \rightarrow \langle \mathtt{m}() \rangle B$. But $\varphi < n + 1 \equiv (\varphi < n \lor \varphi = n)$, so then $\omega \models A \land \varphi < n + 1 \rightarrow \langle \mathtt{m}() \rangle B$. This concludes the inductive case.

$\square$

Back to `fact`. Let's apply this to `fact` from before, and prove that it terminates. One minor annoyance is that $\langle \text{rec} \rangle$ 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 $\langle \text{rec} \rangle$.

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

Recall the contract we stated earlier.

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

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

$$P \equiv \forall x, y. x \geq 0 \land \varphi < n \rightarrow \langle \mathtt{fact}() \rangle y = x!$$

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`.

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

The proof ⓐ corresponds to the branch of the conditional where $x = 0$, and follows easily as seen below:

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

Now we continue with the other branch where the main proof left off. We have to deal with the minor annoyance of the assignment $x := x+1$ occuring before the call to `fact`. We will start the proof off with an application of $[:=]_=$, introducing the fresh variable z into the context to track the value $x-1$. We see that this allows us to cancel the addition of 1 to x after the call, which is as we would expect from our understanding of the program's behavior.

$$
[:=]_=\,\dfrac{=\text{R}\,\dfrac{P, x=n, x>0, u=x-1 \vdash \langle\texttt{fact}()\rangle y*x = x!}{P, x=n, x>0, u=x-1 \vdash \langle\texttt{fact}()\rangle y*(u+1)=(u+1)!}}{P, x=n, x>0 \vdash \langle x:=x-1\rangle\langle\texttt{fact}()\rangle y*(x+1)=(x+1)!}
$$

At this point, we are ready to deal with the call to `fact`. Here is where the fact that we have universally quantified over x, y will help, as we can instantiate x with u to reflect the fact that this is the value the recursive call was given. We will leave y the same.

$$
\forall\text{L}\,\dfrac{=\text{L}\,\dfrac{\to\text{L}\,\dfrac{\mathbb{Z}\,\dfrac{*}{u=x-1, x>0 \vdash u\geq 0 \land u<x} \qquad \text{M}[\cdot]\,\dfrac{=\text{L},=\text{R}\,\dfrac{\mathbb{Z}\,\dfrac{*}{y=(x-1)!, x=n, x>0, u=x-1 \vdash (x-1)!*x=x!}}{y=u!, x=n, x>0, u=x-1 \vdash y*x=x!}}{\langle\texttt{fact}()\rangle y = u!, x=n, x>0, u=x-1 \vdash \langle\texttt{fact}()\rangle y*x=x!}}{(u\geq 0 \land u<x) \to \langle\texttt{fact}()\rangle y=u!, x=n, x>0, u=x-1 \vdash \langle\texttt{fact}()\rangle y*x=x!}}{(u\geq 0\land u<n)\to\langle\texttt{fact}()\rangle y=u!, x=n, x>0, u=x-1 \vdash \langle\texttt{fact}()\rangle y*x=x!}}{\forall x,y.x\geq 0\land\varphi<n\to\langle\texttt{fact}()\rangle y=x!, x=n, x>0, u=x-1\vdash\langle\texttt{fact}()\rangle y*x=x!}
$$

This completes the proof.

6 Summary of today's rules

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

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

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

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

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

References

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