15-414: Bug Catching: Automated Program Verification

Lecture Notes on Recursive Procedures

Matt Fredrikson André Platzer

Carnegie Mellon University

Lecture 12

1 Introduction

Now we've introduced procedures into the language, and seen how to reason about their use modularly using contracts. But so far we've limited ourselves to non-recursive procedures, and although such procedures are widely-used in practice, most languages support recursive calls. As we discussed at the end of the last lecture, recursion introduces significant complications into formal reasoning, as we can no longer understand procedures by reducing them to programs without procedures by inlining. How can we show that a recursive procedure satisfies its contract?

In today's lecture we will focus on this problem. Reasoning about recursion demands that we consider potentially unbounded behaviors. We have dealt with this before, namely when we considered loops and programs with nondeterministic repetition. In that setting, we dealt with the unbounded nature of the verification task using induction, and in particular saw how inductive invariants can be used to write proofs about such programs.

We will make another appeal to induction today, this time to deal with recursive procedures. We will introduce a proof rule that lets us assume that the behavior of a procedure is correct when proving that the procedure body satisfies its contract. However, we need to be careful when using such reasoning to ensure that we are ultimately making an inductive argument; if we do not require that recursive calls operate on "smaller" arguments, then our proofs quickly become unsound. Just as when reasoning about loop convergence, we introduce a variant to show how recursive calls always decrease the context from which they are made. A benefit of this approach is that it allows us to show total correctness, which will be our "default" goal when dealing with recursion.

2 Review: adding procedures

In the last lecture we extended our language with procedure calls. We assumed that the language doesn't have any scoping conventions, so procedures can read and modify any variable in the state. We also assumed that procedures take no arguments, and can modify any variable or array in the state.

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

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

We then introduced some new reasoning principles for procedure calls. Because we assumed that the body of m does not make any recursive calls, we reasoned about calls to m in a straightforward way by simply inlining the body α into the call site.

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

This led to the [inl] axiom.

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

This axiom allows us to reason about program behavior as though the program had originally 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. The [call] rule encapsulates this.

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

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.

3 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; }
}
```

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.

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

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.

Does this allow us to prove `fact` from above? In the following, let

$$Q \equiv \forall n. n \geq 0 \rightarrow [\texttt{fact}()]y = n!$$

Then we can begin our proof as follows.

$$\text{[rec]}\,\dfrac{\text{→R,[if],∧R}\,\dfrac{\text{→R,[:=]}\,\dfrac{\mathbb{Z}\,\dfrac{*}{Q, n = 0 \vdash 1 = n!}}{Q, n \geq 0 \vdash n = 0 \rightarrow [y := 1]y = n!} \qquad Q, n \geq 0 \vdash n \neq 0 \rightarrow [\beta]y = n!}{Q \vdash n \geq 0 \rightarrow [\texttt{if}\,(n = 0)\, y := 1 \,\texttt{else}\, \beta]y = n!}}{\vdash n \geq 0 \rightarrow [\texttt{fact}()]y = n!}$$

Now we need to finish the proof for the "else" branch β, which is where the recursion happens.

$$\text{→R,[;],[:=]}\,\dfrac{\text{[:=]=}\,\dfrac{\text{=R}\,\dfrac{\forall\text{L}\,\dfrac{\text{→L}\,\dfrac{\mathbb{Z}\,\dfrac{*}{n > 0, z = n - 1 \vdash z \geq 0} \qquad \text{M[·]}\,\dfrac{\text{=R,=R}\,\dfrac{\mathbb{Z}\,\dfrac{*}{y = z!, n > 0, z = n - 1 \vdash (n-1)! * n = n!}}{y = z!, n > 0, z = n - 1 \vdash y * n = n!}}{[\texttt{fact}()]y = z!, n > 0, z = n - 1 \vdash [\texttt{fact}()]y * n = n!}}{z \geq 0 \rightarrow [\texttt{fact}()]y = z!, n > 0, z = n - 1 \vdash [\texttt{fact}()]y * n = n!}}{\forall n. n \geq 0 \rightarrow [\texttt{fact}()]y = n!, n > 0, z = n - 1 \vdash [\texttt{fact}()]y * n = n!}}{\forall n. n \geq 0 \rightarrow [\texttt{fact}()]y = n!, n > 0, z = n - 1 \vdash [\texttt{fact}()]y * (z+1) = (z+1)!}}{\forall n. n \geq 0 \rightarrow [\texttt{fact}()]y = n!, n > 0 \vdash [n := n - 1][\texttt{fact}()]y * (n+1) = (n+1)!}}{Q, n \geq 0 \vdash n \neq 0 \rightarrow [n := n - 1; \texttt{fact}(); n := n + 1; y := y * n]y = n!}$$

Great! It looks like this rule will let us prove things about recursive procedures. 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 [AdBO09, Cla77, Cou90, Hes93, Hoa71, vO99]. However, consider the following example.

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

Using [rec] we showed that all terminating runs of `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.

4 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: `abort` $\equiv ?\mathit{false}$, with semantics $[\![\mathtt{abort}]\!] = \emptyset$. To be precise, we define the syntactic approximation $\alpha^{(k)}$ inductively on k as follows:

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

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

We will write $\mathtt{m}^{(k)}$ to denote the procedure obtained by replacing the body with $\alpha^{(k)}$. So, for example applying this to the `fact` procedure from before, $\mathtt{fact}^{(1)}$ 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 $\mathtt{fact}^{(2)}$ 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 $\mathtt{fact}^{(k)}$ 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 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 $\alpha^{(k+1)} = \alpha_{\texttt{m}()}^{\alpha^{(k)}}$. 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.

5 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.

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

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. 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.

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

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

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

and

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

then it is the case that

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

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), for all $n \geq 0$.

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

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

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

The proof proceeds by induction on n.

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

Induction step $n > 0$**.** Fix an arbitrary state ω. The inductive hypothesis gives us that $\omega \models A \wedge \varphi < n \to \langle \texttt{m}()\rangle B$. From this and (4), we have that $\omega \models A \wedge \varphi = n \to \langle\alpha\rangle B$, and by $\langle$inl$\rangle$ that $\omega \models A \wedge \varphi = n \to \langle\texttt{m}()\rangle B$. But $\varphi < n + 1 \equiv (\varphi < n \vee \varphi = n)$, so then $\omega \models A \wedge \varphi < n + 1 \to \langle\texttt{m}()\rangle B$. This concludes the inductive case.

□

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() {
  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:

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

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`.

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

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.

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

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.

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

This completes the proof.

6 Tying it all together

Now that we know ⟨rec⟩ is sound, we can use it to prove that a recursive function satisfies its contract. This was not possible before, for much the same reason that we needed the loop rule to prove postconditions for programs that used $\texttt{while}(Q)\,\alpha$ statements. For nonrecursive procedures, we don't need ⟨rec⟩, because we can reason about the body as though it were its own independent program. Once we have proved the procedure satisfies its contract, we can use ⟨call⟩ when calling the procedure, inserting the contract proof where needed to avoid redundant work. To summarize, we reason about programs with procedures with the following steps.

1. For each procedure `m` that will be invoked, write a contract A, B consisting of a precondition and postcondition, respectively. If we intend to call `m` in contexts requiring proofs for $\Gamma_1 \vdash \langle \texttt{m}() \rangle P_1$, ..., $\Gamma_n \vdash \langle \texttt{m}() \rangle P_n$, then in order to ensure that we can apply ⟨call⟩ to discharge these obligations we need to choose A, B such that there are proofs for $\Gamma_1 \vdash A, \ldots, \Gamma_n \vdash A$ as well as $A \vdash P_1, \ldots, A \vdash P_n$. In other words, we need to be able to prove that the precondition holds each time we call `m`, and that the postcondition is strong enough to let us prove everything we need from our use of `m`.

2. For each procedure `m` with contract A, B, prove that the procedure upholds the contract. That is, assuming that A holds when `m` is called, then B holds when it terminates: $A \vdash \langle \texttt{m}() \rangle B$.

a) If `m` is not recursive, then we can use ⟨inl⟩ to prove this by replacing the call with `m`'s body. After doing so, there will no longer be procedure calls in the program, and we can prove the contract using the same axioms and rules as we would for programs that do not contain procedures.

b) If `m` is recursive, then we must use ⟨rec⟩. This rule allows us to reason about the body of `m` while making the assumption that `m` does in fact satisfy its contract. Each time we encounter a recursive call to `m`, we can apply this assumption to make progress without unfolding the body again, and thus construct a finite proof that the contract holds.

3. Once the contract for each procedure has a proof, use ⟨call⟩ each time a procedure is invoked. This rule requires a proof of $A \vdash \langle\alpha\rangle B$, where α is the body of `m`, but we can supply this with little effort by copying the contract proof from earlier. The remaining obligations are a proof that the precondition holds when the procedure is called, and a proof that the postcondition implies the current goal.

Note that this discussion has focused on proving total correctness. We did not present any reasoning principles for proving partial correctness of recursive procedures. This is by design: now that we know how to prove total correctness, we should always insist on demonstrating correctness with termination.

References

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

[Cla77] Edmund Clarke. Program invariants as fixed points. *1977 Symposium on Foundations of Computer Science*, pages 18–29, 1977.

[Cou90] Patrick Cousot. Handbook of theoretical computer science (Vol. B). chapter Methods and Logics for Proving Programs, pages 841–993. MIT Press, Cambridge, MA, USA, 1990.

[Hes93] Wim Hesselink. Proof rules for recursive procedures. 5, 11 1993.

[Hoa71] C. Hoare. Procedures and parameters: An axiomatic approach. pages 102–116. 1971.

[vO99] David von Oheimb. *Hoare Logic for Mutual Recursion and Local Variables*, pages 168–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.