15-414: Bug Catching: Automated Program Verification

Lecture Notes on Procedures

Matt Fredrikson André Platzer

Carnegie Mellon University

Lecture 11

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.

$$
\begin{array}{lrll}
\text{term syntax} & e, \tilde{e} ::= & x & \text{(where } x \text{ is a variable symbol)} \\
& & \mid c & \text{(where } c \text{ is a constant literal)} \\
& & \mid a(e) & \text{(where } a \text{ is an array symbol)} \\
& & \mid e + \tilde{e} & \\
& & \mid e \cdot \tilde{e} & \\
\text{program syntax} & \alpha, \beta ::= & x := e & \text{(where } x \text{ is a variable symbol)} \\
& & \mid a(e) := \tilde{e} & \text{(where } a \text{ is an array symbol)} \\
& & \mid ?Q & \\
& & \mid \mathtt{if}(Q)\,\alpha\,\mathtt{else}\,\beta & \\
& & \mid \alpha; \beta & \\
& & \mid \mathtt{while}(Q)\,\alpha &
\end{array}
$$

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 $0 \leq i < n$: 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.*

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

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.

4 Summary of today's rules

Today we covered the basics of procedure calls. When a procedure is not recursive, reasoning about the behavior of calls is fairly straightforward by inlining the body into the calling context. The [inl] axiom formalizes this reasoning. We can avoid redundant work by writing contracts, or pre- and postcondition pairs, for procedures. Using the derived rule [call], contracts can be applied at call sites by showing that the context implies the precondition, and the postcondition implies the goal.

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

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

It is also worth noting that both of these reasoning principles have corresponding instances with the diamond modality. The proof of this claim is left as an exercise.

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

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