Formal Methods: Lecture 15

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

An Interpreter

The above functional module has given a precise axiomatization of our chosen subset of Java that is sufficient for reasoning and program verification purposes. But it has done more than that. Since the semantic equations are ground confluent, the above equations have also given an operational semantics to this language.

Indeed, we can describe the execution of the language by algebraic simplification with the equations from left to right.

Therefore, the above equations have in essence given us an interpreter for our language.

An Interpreter (II)

As a reminder, you can create an initial state by just writing `initial`, you can then execute some code, say `bs` from that state with `initial | bs`.

Make sure to use parentheses around `bs`, otherwise there will be parsing problems. If you now don't want to see the complete resulting state but only the value of a specific variable `'x` then `(initial | bs)['x]` will do that for you.

Mind that this gives you a value, so if you expect to see a 42 you will get a `int(42)` which is the integer value used by our language.

An Interpreter (III)

Some examples are:

```
red (initial | (int 'x = #i(5) ; int 'y = #i(4) ;
                {'x = #i(42) ; } 'x = 'x + 'y ;))['x] .

red (initial | (int 'x = #i(5) ; int 'y = #i(4) ;
                {int 'x = #i(42) ; } 'x = 'x + 'y ;))['x]  .
```

which return

```
rewrites: 86 in 10ms cpu (10ms real) (8600 rewrites/second)
result Value: int(46)
```

respectively, because of shadowing of the assignment to `'x` in the block,

```
rewrites: 86 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(9)
```

An Interpreter (IV)

```
red (initial | (int 'x = #i(0) ; if #b(false)
                          if #b(true) 'x = #i(1) ;
                        else 'x = #i(2) ;))['x]  .
```

This is interesting because it shows us how the dangling else problem is solved in this particular implementation.

According to the Java Language Specification an else block belongs to the innermost if part, unless parentheses show otherwise. The result in this case is:

```
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(0)
```

It would have been `int(2)` if the else was attributed (wrongly) to the outer if.

An Interpreter (V)

```
red (initial | (int 'a = #i(3) ; int 'b = #i(2) ; int 'c = #i(1) ;
                int 'd = 'a - 'b + 'c ;))['d] .
```

With this example, which returns

```
rewrites: 241 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(2)
```

we have shown the left-associativity of arithmetic operators.
With right-associativity the result would have been 0.

An Interpreter (VI)

```
red (initial | (int 'x = #i(7) ; int 'y = #i(5) ;
                int 't = 'x ; 'x = 'y ; 'y = 't ;))['x] .
red (initial | (int 'x = #i(7) ; int 'y = #i(5) ;
                int 't = 'x ; 'x = 'y ; 'y = 't ;))['y] .
```

A simple swap example indeed swaps the values of `'x` and `'y`, the results are 5, respectively 7, as expected.

```
red (initial | (int 'n = #i(5) ; int 'c = #i(0) ; int 'x = #i(1) ;
                    while ('c < 'n) { 'c = 'c + #i(1) ; 'x = 'x * 'c ; })) ['x] .
```

Finally this computes the factorial of 5 and thus returns:

```
rewrites: 416 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(120)
```

Formal Reasoning

In fact, the equational axiomatization of our language is much more than an interpreter.

Since we have axiomatized the language as an equational theory, we can do inference about our programs using such a theory.

In fact, since the semantic equations are ground-confluent, such equational reasoning is mechanically supported by the Maude system. Therefore, we can do a substantial amount of formal reasoning about programs in our language, even without using induction.

Formal Reasoning (II)

We may reason about some properties of a given program. In general, of course, such reasoning may require induction, but in some cases plain equational deduction will deliver the goods.

Consider, for example, the following informal specification of a swap program, swapping the values of, say, variables `'X` and `'Y`, namely, that for any state the values in `'X` and `'Y` should have been swapped after execution.

Formal Reasoning (III)

Actually we cannot do that for any state but only for states in which `'X` and `'Y` are already declared. Such a state is created by `ctxState((int 'X ; 'int 'Y ;))`. The fact that this represents any state with `'X` and `'Y` declared results from the generic values put into both variable's locations.

Whether other variables are declared or not does not matter, as long as a candidate for the swap program does not use other variables without declaring them first. This is verified with a quick look at the semantics.

Formal Reasoning (IV)

This correctness specification can be made mathematically precise as the equations:

$$\texttt{(ctxState((int 'X ; int 'Y ;)) | (swap))['X]} = \texttt{(ctxState((int 'X ; int 'Y ;)))['}$$

$$\texttt{(ctxState((int 'X ; int 'Y ;)) | (swap))['Y]} = \texttt{(ctxState((int 'X ; int 'Y ;)))['}$$

Consider now a possible candidate for our `swap` program, namely the program,

```
int 'T = 'X ; 'X = 'Y ; 'Y = 'T ;
```

Formal Reasoning (V)

We can verify that this program meets its specification just by equational simplification as follows,

```
reduce in JAVAX :
ctxState((int 'X ; int 'Y ;)) | (int 'T = 'X ; 'X = 'Y ; 'Y = 'T ;)['X]
== ctxState((int 'X ; int 'Y ;))['Y] .
rewrites: 112 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in JAVAX :
ctxState((int 'X ; int 'Y ;)) | (int 'T = 'X ; 'X = 'Y ; 'Y = 'T ;)['Y]
== ctxState((int 'X ; int 'Y ;))['X] .
rewrites: 112 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Preconditions

Recall the specification of correctness for the program `swap` to swap the value of variable `'X` and variable `'Y` that we have already discussed. With S taking the place of `ctxState((int 'X ; int 'Y ;))`, but with the universal quantification implicitly limited to states of that form we get the specification,

$$(\forall \texttt{I} : \texttt{Int})(\forall \texttt{J} : \texttt{Int})(\forall \texttt{S} : \texttt{State})\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}$$

$$\Rightarrow \texttt{(S | (swap))['X]} = \texttt{int(I)} \wedge \texttt{(S | (swap))['Y]} = \texttt{int(J)}$$

Here we have the implicit precondition that we are starting in a state where `'X` and `'Y` are declared as described above.

Preconditions (II)

We shall call the equation

$$\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}$$

which is assumed to hold before the execution of the program, the precondition.

Note that this equation has a single occurrence of the state variable S in each equation, and can be thought of as a state predicate, having the integer variables I and J as parameters.

Postconditions

Consider in the above specification the equation

$$\texttt{(S | (swap))['X]} = \texttt{int(I)} \wedge \texttt{(S | (swap))['Y]} = \texttt{int(J)}$$

which is supposed to hold after the execution of a program. This can also be viewed as a state predicate, namely the state predicate

$$(\dagger) \quad \texttt{(S)['X]} = \texttt{int(I)} \wedge \texttt{(S)['Y]} = \texttt{int(J)}$$

applied not to `S`, but instead to the state `S | (swap)` after the execution. We call $(\dagger)$ the postcondition. Note that it also has the integer variables `I` and `J` as extra parameters.

State Predicates

This example suggests a general notion of "state predicate," intuitively a property that holds or doesn't hold of a state, perhaps relative to some extra data parameters. Since in our language the only data are integers, such parameters must be integer variables.

Therefore, for our language we can define a state predicate as a conjunction of equations

$$t_1 = t'_1 \ \wedge \ldots \wedge t_n = t'_n$$

in a module extending `JAVAX`, such that the set V of variables in all terms in the equations has at most one variable `S` of sort `State`, which possibly appears more than once, and the remaining variables are all of sort `Int`.

State Predicates (II)

How general is this definition of state predicate?

One can of course generalize things further, by allowing an arbitrary first-order formula (with the same condition on its variables V) instead of just a conjuction of equations. Also, the notion extends naturally to other sequential languages which may have other data structures besides integers.

However, in practice the notion is quite general; among other things because, using an equationally defined equality predicate, we can express arbitrary Boolean combinations of equations as a single equation.

State Predicates (III)

Note that, although a state predicate has at most one variable, say `S`, of sort `State`, it is perfectly possible for `S` to be mentioned more than once. For example,

```
S['X] = S['Y]
```

is a perfectly acceptable state predicate; and `S` could likewise appear in several conjuncts of a state predicate.

Hoare Triples

The above example of our specification for `swap` is paradigmatic of a general way of specifying properties of a sequential imperative program p by means of a Hoare triple (after C.A.R. Hoare)

$$\{A\}\, p\, \{B\}$$

where A and B are state predicates, called, respectively, the precondition, and postcondition of the triple.

In this notation, the specification of `swap` becomes rephrased as,

$$\{\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}\}$$

$$\texttt{swap}$$

$$\{\texttt{(S)['X]} = \texttt{int(I)} \wedge \texttt{(S)['Y]} = \texttt{int(J)}\}$$

Hoare Triples (II)

Given our algebraic approach to the semantics of imperative programs, this is all just an (indeed very useful) *façon de parler* about an ordinary first-order property satisfied by the initial model of our language, namely the initial algebra T_{JAVAX}.

Therefore, we define the partial correctness of a program `p` with respect to a Hoare triple by the equivalence,

$$T_{\text{JAVAX}} \models \{A\}\ \texttt{p}\ \{B\} \quad \Leftrightarrow$$

$$T_{\text{JAVAX}} \models (\forall V)\ A \wedge\ (\texttt{(S | p) : WrappedState}) \Rightarrow (B(\texttt{S/S | p})).$$

Note that `p` terminates iff `(S | p) : WrappedState`.

Hoare Triples (III)

In our `swap` example this becomes,

$$T_{\texttt{JAVAX}} \models (\forall \texttt{I} : \texttt{Int})(\forall \texttt{J} : \texttt{Int})(\forall \texttt{S} : \texttt{State})\,\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}$$
$$\wedge\ \texttt{(S | swap) : WrappedState}$$
$$\Rightarrow \texttt{(S | (swap))['X]} = \texttt{int(I)} \wedge \texttt{(S | (swap))['Y]} = \texttt{int(J)}.$$

which is just our original correctness condition with the addition of the termination condition `(S | swap) : WrappedState`. What we mean by partial correctness, as opposed to total correctness is that termination, rather than being always required, becomes an additional assumption needed for the postcondition to hold.

Of course, since `swap` was a terminating program, this was superfluous in that case, but it is not superfluous when iterations are involved.

Compositionality: Hoare Logic

An important contribution of Hoare was to propose his triples as a compositional logic of programs, by giving a collection of inference rules based on the structure of the program text to decompose proof of correctness of more complex programs into proofs for simpler subprograms.

For example, to prove the correctness of a sequential composition `p q` he gave the rule,

$$\frac{\{A\}\ \mathtt{p}\ \{B\} \qquad \{B\}\ \mathtt{q}\ \{C\}}{\{A\}\ \mathtt{p}\ \mathtt{q}\ \{C\}}$$

which can be easily justified by analyzing the semantics of the triples.

Compositionality: Hoare Logic (II)

Also a very useful rule, of easy justification, is the consequence rule,

$$\frac{A \Rightarrow A_1 \quad \{A_1\}\ \mathtt{p}\ \{B_1\} \quad B_1 \Rightarrow B}{\{A\}\ \mathtt{p}\ \{B\}}$$

Another rule of easy justification is the rule for the skip program ;, that, rephrased in our terms, takes the form,

$$\frac{}{\{A\}\ ;\ \{A\}}.$$

Compositionality: Hoare Logic (III)

All the Hoare rules mentioned so far are quite generic, in the sense that they will apply to many languages. But note that there isn't a single Hoare logic: different programming languages may have somewhat different Hoare rules, depending on their semantics.

In our Java fragment the fact that tests may have side effects makes it necessary to give slightly more subtle Hoare rules for conditionals and loops than those originally proposed by Hoare for those two constructs, because Hoare did not contemplate side-effecting tests in his original language.

Compositionality: Hoare Logic (IV)

Indeed, for conditionals we need to work a little harder. Here, `evalTst(S, TE)` gives the boolean which the evaluation of the test expression `TE` in state `S` returns. With `|` we create an operator which separates the "execution" of a test expression from the rest of the program. It is not possible to use the usual concatenation __ because the test expression is not of the same sort, but using the `|` operator it can be evaluated first, so its side effects change the state, and then the result gets thrown away and the execution continues as usual.

$$\frac{\{A \wedge (\texttt{evalTst(S, t)} = \texttt{true})\}\ \texttt{t | p}\ \{B\} \quad \{A \wedge (\texttt{evalTst(S, t)} = \texttt{false})\}\ \texttt{t | q}\ \{B\}}{\{A\}\ \texttt{if t p else q}\ \{B\}}$$

Compositionality: Hoare Logic (V)

The above captures the usual semantics of `if`, just as in simpler languages, but in contrast here we have this `t |` in front of the execution of the two branches of the conditional in the respective cases. It is not enough here to know that `t` evaluated to either true or false, which is what the two properties assure, but also `t` needs to have been executed for its possible side effects to take effect and change the state. This Hoare rule still simplifies things as we now do not have to take a decision based on the test value anymore but we just have to have the test expression executed. One could also give a sequential composition rule for | additionally. The extra effort here is necessary because of side effects!

The Loop Rule and Invariants

A very important rule is the proof rule for the partial correctness of while loops. Here we face the same problem as with conditionals because the loop condition can also have side effects. It takes the form,

$$\frac{\{A \wedge (\texttt{evalTst(S,t)} = \texttt{true})\}\ \texttt{t}\ \texttt{|}\ \texttt{p}\ \{A\} \quad \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}\ \texttt{t}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}}{\{A\}\ \texttt{while t p}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}}$$

It requires a somewhat more involved justification. The state predicate A is called an invariant of the loop. Note that, by the partial correctness interpretation of Hoare triples, the holding of the postcondition implicitly depends on the termination assumption for the loop.

Factorial Example

Consider again our factorial program. To prove its correctness, intuitively that it correctly computes the factorial function, we first need to define mathematically such a function, by adding to `Int` an error supersort `Int?` of `Int`, a function,

```
op _! : Int -> Int? .
```

and equations,

```
var I : Int .
eq 0 ! = 1 .
ceq I ! = I * (I - 1)! if 0 < I .
```

Of course, the factorial function is not defined for negative numbers, so that a program may not even terminate if the original input is negative.

Factorial Example (II)

Therefore, we should give the requirement that the input variable `'N` is nonnegative as a precondition, yielding the specification,

$\{(\texttt{S['N]} = \texttt{int(I)}) \wedge (\texttt{0 <= I} = \texttt{true})\}$ `factp` $\{\texttt{S['X]} = \texttt{int(I !)}\}$.

The above specification takes the point of view of a customer who specifies properties of the desired program. An implementer may then give to the customer the following `factp` program:

```
'C = #i(0) ; 'X = #i(1) ; while ('C < 'N) { 'C = 'C + #i(1) ; 'X = 'X * 'C ; }
```

Factorial Example (III)

The question, then, is how to prove this program correct. To do so we can:

- use the Hoare logic rules, which we have justified, and
- use inductive reasoning, since the correctness of Hoare triples reduces to satisfaction of first-order formulas in the initial model $T_{\texttt{JAVAX}}$.