Project 3: Bytecode Verification Engine

Program Verification Course 12/13

1 Setup

In this assignment we will implement a Weakest Pre-condition calculus which will enable you to do automated verification. With a little tweak, the same feature will also enable you to automatically generate test-cases.

As a starting point we will consider a simplified byte-code like language, which we will call $Lang_0$. However, you still get the usual structured programming's control structures, so that you do not have to worry about dealing with jump instructions that you normally have in many byte-code languages.

Later, you can earn more points by implementing more sophisticated extentions to the language.

Here is an example of our base language $Lang_0$:

```
prog P(2) {    -- declaring 2 parameters
  local 1 ;    -- declaring 1 local var.
  SETLOCAL 0 10 ;  -- local-var0 := 10
  PUSHPARAM 0 ;      -- push arg0 to the stack
  PUSHPARAM 1 ;
  ADD ;            -- add two top values in the stack
  LOADLOCAL 0 ;    -- load local-var0 to the stack
  EQ ;             -- compare if two top values in the stack are equal
  iftrue { PUSHLITERAL  1 ; } -- if then-else, based on top of the stack
  else   { PUSHLITERAL -1 ; }
  return ;
}
```

This defines a program called `P` that takes two parameters and has one local variable. In a higher level language, it is equivalent to:

```
prog P(a1,a2) {
  var x0 = 10 ;
  if (a1+a2 == x0) { return 1 ; }
  else { return -1 ; }
}
```

Program A *program* has a name, and specifies the number of parameters and local variables it has. The body of a program is a *statement*

Statement A *statement* is one of these:

1. An *instruction*.
2. `iftrue` $statement_1$ `else` $statement_2$. The top of the stack is poped out. If the value is *true*, $statement_1$ will be executed, and else $statement_2$.
3. A list of *statements*.

Instruction Available instructions are:

1. `SETLOCAL` k lit. Set the value of the k-th local variable to the literal lit.
2. `LOADLOCAL` k. Push the value of the k-th local variable to the stack.
3. `STORELOCAL` k. Pop the stack, and put the value in the k-th local variable.
4. Similarly we have `LOADPARAM` k and `STOREPARAM` k to push and store to the k-th parameter of a program.
5. `PUSHLITERAL lit` push the literal lit to the stack.
6. `POP` to pop the stack. The poped value is thrown away.
7. `ADD`, remove the two top values on the stack, add them, and put the result back at the top of the stack. Similarly you have `MIN`, `MUL`, `LT`, `LTE`, `GT`, `GTE`, `EQ`.
8. `return` will return from the current program. The value at the top of the stack is assumed to contain the return-value.
   This instruction can only appear as the *last* instruction in a program.

Data types We will only have integer and boolean types. The integer type is assumed to have infinite range.

Specification A program can be specified by a pre- and post-conditions, e.g.:

$$\{a_0 > a_1\} \;\; P(2) \;\; \{(\texttt{return} = -1) \vee (\texttt{return} = 1)\}$$

We use a_k to refer to the k-th parameter of P. However, if it appears in the post-condition, it refers to the old value of a_k, which is its value at the beginning of P. So, for example:

$$\{true\} \;\; P(2) \;\; \{(a_0{+}a_1 = 10) = (\texttt{return} = 1)\}$$

says that P's return value is 1 if and only if the *original* $a_0 + a_1$ is 10.

Simple expressions as well as first order $\forall$ and $\exists$ quantifications can be used in a specification. E.g.:

$$\{true\} \;\; isDivBy(2) \;\; \{\texttt{return} = (\exists k : int.\ a_0 = k * a_1)\}$$

In the post-condition you should not refer to the stack.

2 Base Task (Mandatory, 7pt)

Come up with a weakest pre-condition calculus for $Lang_0$ then build a verification system for $Lang_0$ based on the calculus. You can implement in any language –I recommend Haskell. You can use any verification back-end –I recommend Microsoft's Z3 theorem prover.

Use the calculus to verify $Lang_0$ programs. Consider a specification:

$$\{p\} \;\; P(k) \;\; \{q\}$$

Let w be the weakest pre-condition of $P(k)$ with respect to the post-condition q. The following are equivalent:

1. The above specification is valid.
2. $p \Rightarrow w$ is valid.
3. $\neg(p \Rightarrow w)$ is unsatisfiable.

The last two statements can be checked with an automated theorem prover like Z. We can of course only automatically verify formulas which are within the decidability range of your back-end prover.

Don't bother with concrete parsers. We will only consider type correct instances of $Lang_0$. For demonstration, it is sufficient to provide examples in some internal representation that are reasonably readable. Furthermore, we assume our target programs to have been statically checked:

1. They are type correct.
2. They contain no reference to undeclared parameters or local variables.

2.1 Representing Stack

You need to keep in mind that the back-end theorem prover may not have 'stack' as its primitive data type. However, e.g. in Z3 you can logically represent an array a as function e.g. $a : Int \to Int$. It takes an index, and returns the content of the array at that index.

Another complication is that you want to push both integers and boolean values to the stack. So the representation $a : Int \to Int$ will not do. You can either choose to represent boolean values as integers, or define a custom datatypes similar to *Either* if it is supposted by the theorem prover (it is supported by Z3, but I don't know how clever it subsequently handles verification involving it).

2.2 To deliver

- Your implementation.
- Provide an accompanying report, listing and explaining your calculus.
- Provide two examples demonstrating your verification tool. Explain these examples in the report as well.

3 Extensions

Below are some extensions to challange you. Doing them will give you more points. You can do any number of extensions. The maximum total grade is 10. You can also propose your own extensions, but you need to discuss them with me first.

For each extension that you do, you need to:

- Extend your report by explaining the new part and the modification of the weakest pre-condition calculus that you need to implement the extension.
- Provide two examples demonstrating the extension. Explain these examples in the report as well.

3.1 Return from anywhere, 1pt

Let's drop the restriction that `return` should be the last instruction in a program. You will have to extend your calculus to accomodate this.

3.2 Automated test-cases generator, 1pt

Extend $Lang_0$ with a new statement `MARK`. It does nothing except to place a 'marker' at the statement's location. I leaves it to you to define how the marking is concretely done.

Let $P(k)$ be a $Lang_0$ program that contains at least one `MARK` statement. An execution of $P(k)$ is said to be *positive* if it passes at least one marked position. A test-case is a set of values for P's parameters that will cause P to execute positively.

Let w be the weakest pre-condition of $P(k)$ so that it executes positively. You can use a theorem prover to check w's satisfiability. Any instantiation satisfying it is essentially a test-case.

Extend your verification tool with a feature to automatically generate test-cases.

Dealing with loops with bounded verification, 2pt

Extend $Lang_0$ with a new statement `whileTrue` S. (1) It pops the top of the stack. (2) If the popped value is true, it will execute the statement S and then go back to step (1). Else, the whole `whileTrue` statement is finished, and we proceed with the next statement.

For later, let us also add virtual instructions `TOPTRUE` that puts a constraint that the top of the stack at that moment must be true.

In general it is not possible to calculate the weakest pre-condition of a loop. In Hoare logic (to be dicussed in the Lectures), you can annotate a loop with an invariant I, which then we can take as the loop's 'weakest' pre-condition[1].

Let us here do a different approach that does not require the programmer to come up with such an invariant. The approach is however *incomplete* in the sense that if the verification finds an error, then it will be a real error. However, a positive verification result does not exclude all errors.

In *bounded verification* we only verify executions up to some given length N. To measure the length of execution let us just count the number of instructions and guard evaluations (in `iftrue` and `whileTrue`) in the execution.

Consider a program $P(k)$ that contains a loop `whileTrue` S and a post-condition q of the loop. Let w_k be the weakest pre-condition so that the loop iterates exactly k-times and then terminates in Q. This w_k can be calculated as follows:

$$\begin{array}{lcl} w_0 & = & \neg stack_{top} \ \wedge \ \mathsf{wp}\ \texttt{POP}\ q \\ w_{k+1} & = & stack_{top} \wedge \mathsf{wp}\ \{\texttt{POP};\ S\}\ w_k \end{array}$$

The weakest pre-condition of the loop is the infinite disjunction $w_0 \vee w_1 \vee w_2 ...$ (which as pointed out, cannot be in general calculated). But in bounded verification you only take those w_k so that the overall length of the execution of P does not exceed N steps.

Extend your verification tool to support bounded verification and is able to deal with loops in that way.

Exception, 2pt

Extend $Lang_0$ with a new statement `DIV`, that will: (1) pop the two top values from the stack, (2) divide the first with the second, (3) put the result back on the stack. Performing a division by 0 will throw an exception.

Furthermore, every instruction and statement is now labelled by its 'line number'. The structure of programs is extended so that it also has a list/table H of

[1] It is actually not the weakest one. It is a sufficient one to guarantee that when the loop terminates, it will terminate in I itself.

exception handlers. This list can be empty. Each element H is called a *handler*, which has this structure:

$$(begin, end, h)$$

where *begin* and *end* are natural numbers, and h is a statement.

Let $P(k)\ S\ \triangleright\ H$ be a program with S as its normal body, and H as its list of exception handlers. An execution of P now proceeds in one of the following ways:

1. S does not throw an exception. Then P ends normally, in a state that would be marked as a *normal state*.
2. An instruction or a statement at line n throws an exception, we look for the first handler (b, e, h) such that n falls within 'its range'. That is, $b \leq n < e$. If such a handler can be found, the execution continues with h. There are now two possibilities:
   (a) h does not throw an exception. When h ends, P also ends. The final state is a normal state.
   (b) h throws an exception. This ends h, and also ends P. The final state is marked as an *exceptional state*.
3. No handler can be found to handle the 2nd situation above. Then p ends, and the final state is an exceptional state.

The concept of 'post-condition' is now extended to a pair (Q_n, Q_e) where Q_n specifies the required post-condition when a program ends normally, and Q_e for when it ends by an exception.

4 Some hints

Below T represents the pointer to the top of the stack. It is initialized by -1. So, in a stack with one element, its top element is $stack_0$ and $T = 0$. Some possible wp-rules for some constructs of $Lang_0$ are shown below as examples. These rules assume that in the specifications (pre/post-conditions) you do not quantify over the stack's elements, nor do you use a variable that can explicitly point to an arbitrary element of the stack.

1. $wp\ (\texttt{SETLOCAL}\ k\ x)\ Q \quad = \quad Q[x/loc_k]$
2. $\texttt{LOADLOCAL}\ k \ = \ \{T := T{+}1\ ;\ stack_T := loc_k\}$. So the wp is:

   $$Q[loc_k/stack_T][T{+}1/T]$$

3. $\texttt{ADD} \ = \ \{stack_{T-1} := stack_{T-1} + stack_T\ ;\ T := T{-}1\}$. So the wp is:

   $$T \geq 1\ \wedge\ Q[T{-}1/T][stack_{T-1} + stack_T/stack_{T-1}]$$

5 wp of the example program

Consider this (invalid) specification:

$$\{a_0 > 0\} \;\; P(2) \;\; \{(a_0 = a_1) = (\texttt{return} = 1)\}$$

This should give this wp (after simplification):

$$(a_0 = a_1) \;=\; (a_0{+}a_1 = 10)$$

Depending on the specifics of your wp-rules you may get a different formula, e.g. you may additionally explicitly require that the stack should be in a valid state.

In any case, assuming the above wp, the validity of the specification is thus equivalent to:

$$a_0 > 0 \;\Rightarrow\; ((a_0 = a_1) \;=\; a0{+}a1 = 10)$$

which is satisfiable, but not valid. So, the specification is not valid as well.