Exercises in Symbolic Execution

Uppsala University, Spring 2021

Exercise 1: Constraint Solving

In this exercise we will extend the nano-symex/dse implementation from the lectures, so that also programs operating on arrays can be analysed. The implementation can be found on github, https://github.com/pruemmer/nano-symex. For the exercise, we recommend that you create a fork of the repository, in which you can then add your changes. The exercise is supposed to be solved **individually** by students attending the PhD course.

If you are unsure about Scala programming, you will find a good introduction in the book

> "Programming in Scala", Odersky, Martin; Spoon, Lex; Venners, Bill

You should also be able to find pdfs of this book online.

The goal of the exercise is to implement two different approaches to handle arrays in symbolic execution:

(A) A direct representation using the SMT-LIB theory of extensional arrays. In this version, every program array is directly mapped to an array handled by the constraint solver. Reading from an array is implemented using the SMT-LIB `select` function, and updating arrays using the `store` function. Required case splits (checking whether two array indexes coincide) are entirely handled by the SMT solver.

(B) Finite modelling of arrays by storing the (finitely many) elements of an array that have already been accessed, using scalar solver variables. In this version, every array access (read or write) has to check whether the accessed index corresponds to an array element that has been accessed before. If that is the case, the solver variable modelling this array element will be read or updated; if the array index has not previously been accessed, a new solver variable is introduced and initialised non-deterministically. This encoding avoids the use of SMT-LIB arrays, but introduces additional branching when arrays are accessed at symbolic (non-concrete) indexes.

We only consider the case of arrays storing integer numbers, and assume that all arrays have unbounded size.

Example Program: Insertion Sort

After extending the implementation, you should be able to analyse the following program using approaches (A) and (B) (for inputs of fixed length len), and compare the performance of the two methods:

```
Inputs: array A, array length len

i ← 1
while i < len
   x ← A[i]
   j ← i - 1
   while j >= 0 and A[j] > x
      A[j+1] ← A[j]
      j ← j - 1
   end while
   A[j+1] ← x
   i ← i + 1
end while

// Assertion: the final array is sorted
i ← 0
while i+1 < len
   assert(A[i] <= A[i+1])
   i ← i + 1
end while
```

The following steps will guide you through the process of extending the symbolic execution engine.

Step 1: SMT Support for Arrays [optional]

Our SMT solver interface already supports the theory of arrays. To introduce an array variable, simply a variable of the type `(Array Int Int)` is declared. Elements in the array can then be accessed using the functions `select` and `store`. Most solvers in addition support the function `((as const (Array Int Int)) <value>)` to create constant arrays, i.e., arrays that store the same value at all positions. Constant arrays will not be needed for our symbolic execution engine, however.

There is one point not yet supported in the SMT layer: the extraction of solutions when arrays are involved. This functionality is not needed for plain symbolic execution, but required for dynamic symbolic execution.

Task: Add a method for extracting the value of an array variable to the `SMT` class.

```
[...]
def getArrayValue(name : String) : Map[BigInt, BigInt]
[...]
```

Step 2: Syntax for Arrays in Programs

We need to be able to express programs operating on arrays, for instance the insertion sort example. The implementation on https://github.com/pruemmer/nano-symex already provides an array type, but no operations for accessing arrays yet.

Tasks:

1. Add a binary operation `ArElement` for reading the value of an array at a given index. This operation corresponds to a new sub-class of `Expr`. You can optionally define syntactic sugar for array accesses, have a look at the methods `If`, `While`, `ite2RichIte`, etc.
2. Extend the `Assign` statement (sub-class of `Prog`) to also support assignments to arrays. This means that `ArElement` has to be supported as the left-hand side of assignments. The `Assign` class can already handle this case, but a method corresponding to `var2LHS` has to be added.
3. Translate the insertion sort program to our program AST.

Step 3, (A): Symbolic Execution using SMT-LIB Arrays

We will now extend our implementation to support symbolic execution using method (A). This requires us to touch two of the classes of our implementation, the `ExprEncoder` class for

translating program expressions to solver expressions, and the `SymEx` class for the actual symbolic execution.

Tasks:

1. Extend `ExprEncoder` and `IntExprEncoder` to handle also array variables. This means that, in addition to `IntType`, also a translation of the array type has to be defined, and the `encode` methods have to be extended to handle `ArElement`.
2. [Optional] Extend also the `eval` methods to handle arrays. This first requires a more general definition of the `Valuation` type.
3. Extend the `SymEx` class to handle array variables. This means that also solver variables for arrays have to be declared, and assignments to `ArElement` expressions have to be mapped to the SMT-LIB function `store` (in the recursive method `execHelp`).
4. Test your implementation using the insertion sort program. Note that symbolic execution will not terminate in general, since the program contains an unbounded loop; to make the analysis terminating you can fix the input variable “len” to some concrete value. What is the size of the biggest (symbolic) array you can sort with a 60s timeout?

Step 4, (B): Symbolic Execution using Finite Modelling of Program Arrays

We now consider symbolic execution with the method (B). This approach completely hides arrays from the solver, and instead models arrays as a set of scalar solver variables. As a result, the implementation is a bit more involved than the one of (A). As a simplifying assumption, we only consider programs that access arrays using statements of the form:

```
x := ArElement(...)          // Reading from an array (1)
```

or

```
ArElement(....) := t         // Writing to an array (2)
```

Note that every program can be rewritten to only access arrays using statements of the form (1) or (2).

Tasks:

1. Implement a function for normalising programs to only use array statements (1) or (2). Alternatively, you can also rewrite the insertion sort program by hand.
2. Add a *symbolic array store* to the `SymEx` class. This array store can be passed along together with the existing symbolic store, and maps every array variable to a set of scalar variables. The type of the symbolic array store can be

   ```
   Map[Var, Map[BigInt, String]]
   ```

   so that every array variable is defined by a finite set of solver variables. For instance, the map

   ```
   Map(x -> Map(0 -> ”x0”, 10 -> ”x10”))
   ```

expresses that the array variable `x` is symbolically defined at the indexes 0 and 10, and the value at those indexes is represented by the solver variables `x0` and `x10`, respectively. The value of `x` at other indexes is not modelled yet.

3. The most complex part: extend the recursive `execHelp` function to support reading from arrays (1) and writing to arrays (2). In both cases, the accessed index of the array has to be determined, and in general the constructed path constraint can allow multiple possible indexes. This case has to be modelled as separate execution paths.
   There is a simple schema for implementing such branching: when symbolically executing statements (1) or (2), ask the SMT solver for the value *v* of the accessed array index `ind`, using the method `getSatValue`. Generate then two execution branches: on the first branch, add the path constraint `ind == v`, and continue symbolic execution assuming that the array is accessed at index *v*; on the second branch, add the constraint `ind != v`, and recurse. [Various optimisations are possible.]
   Once *v* has been determined as the accessed array index, look up the symbolic value of the array at this point in the symbolic array store. If the array store does not contain an entry for this index yet, declare a new solver variable and add it to the array store.
4. As in Step 4, test your implementation using the insertion sort program, and the size of the biggest (symbolic) array you can sort with a 60s timeout. Which method turns out to be more efficient, (A) or (B)?

Step 5: Extend the Dynamic Symbolic Execution Algorithm to handle Arrays as well [optional]

Submission

Write a short report (2-3 pages) describing your implementation, and comparing the performance you observed for methods (A) and (B). Include a link to the source code of your implementation.

Exercise 2: KLEE

[upcoming]