CS 766: Assignment

Submitted by:

Debabrata Mallick (203051004)

Submitted to:

Ashutosh Gupta

Introduction

Our task was to develop a tiny software tool that takes a program as input and outputs all valid traces.

Input program is allowed to have assert statement in it. If there exists a sequential execution(valid trace) that violates the assert statement, then our tool stops exploring traces and outputs the result as "Assertion violation" and displays this trace.

Input constraints:

1. 1 ≤ n ≤ 10.
2. Maximum number of instruction per process = 4
3. Fixed global variables: x , y, and z.
4. Maximum number of local registers in the input program= 10
5. Each read instruction can obtain its value from max 4 write instruction. So, if there is read instruction, r = x, then it can obtain a value of x from max 4 write instructions on variable x.
6. Initially x=0 , y=0 , z=0. (Global variables are initialised with zero.)

Our approach

We take the inputs according to input constraints which we further modify to get them in a workable form. We permute the instructions to get all possible concurrent program executions. Going further we take each permutation and check its *traces*, _rf _and *ws*. We collect *trace*, _rf _and _ws _for each execution. We collect only the unique traces in terms of combination of read from(rf) and write serialization(ws) relations.

If an assert conditional statement is present then we check its validity against the current concurrent execution. If it is valid then we move on to next concurrent execution else we throw an assert violation error and output the *trace*, _rf _and _ws _for current concurrent execution and exit the program.

If the program doesn't contain an assert statement or all traces seem to satisfy the assert statement then we print the possible *trace*, _rf _and *ws*.

```
def main():
    statements, assert_cond_stat = inputProgram()
    statements, all_variables_dict = modifyInput(statements)
    initial_stat = statements[0]
    statements = statements[1:]
    perm_of_statements = permStatements(statements)

    traces = []
    rfs = []
    wss = []
    rfs_wss = []

    for perm_of_statement in perm_of_statements:
        trace = None
        rf = None
        ws = None
        trace = getTrace(perm_of_statement)
        rf = getRF(perm_of_statement)
        rf.sort()
        ws = getWS(perm_of_statement)

        if(assert_cond_stat != None):
            flag  =
checkAssert(perm_of_statement,all_variables_dict,assert_cond_stat)
            if(not flag):
                assertViolateOutput(trace,rf,ws)
                sys.exit()

        temp = rf+ws
        if(temp in rfs_wss):
            continue

        traces.append(trace)
        rfs.append(rf)
        wss.append(ws)
        rfs_wss.append(temp)

    output(traces,rfs,wss)
```

How to Use

Just run the file trace.py in command line `python3 tarce.py,` or by using any python IDE like pycharm and hit the run button after opening the trace.py file. You will get instruction for providing inputs.

Taking Inputs

We created the `inputProgram()` function which takes input as per the input constraints specified in the problem statement.

```
def inputProgram():

    print("Provide number of processes:")
    n = int(input())
    if(n<1 or n>10):
        print("Value of n should be between 1 and 10 inclusive.")
        sys.exit()
    programs = [None] * (n + 1)
    programs[0] = "x=0;y=0;z=0;"

    for i in range(1, n + 1):
        print("Type the program " + "P" + str(i))
        programs[i] = input()

    print("Is there any assertion statement, if yes type 1 else 0")

    assert_cond_stat = None
    if (int(input()) == 1):
        print("Type assertion conditional statement:")
        assert_cond_stat = input()
        assert_cond_stat = assert_cond_stat.replace("assert","")

    else:
        assert_cond_stat = None

    statements = []
    for program in programs:
        temp = program.split(';')
        temp = temp[:-1]
        if(len(temp) > 4):
            print("Maximum number of instructions in the program cannot be more
than four.")
            sys.exit()
        statements.append(temp)

    return statements, assert_cond_stat
```

First line of input is the number of programs, n. Then the next n lines contain instructions from each of the programs which are stored in the `programs` list. Then in the next line we ask whether the next line will contain an assert statement, if input is 1 then we take the assertion control statement as input in `assert_cond_stat`. If the input is 0 then there will be no assert statement for the given program. We then check if each program contains ≤ 4 instructions per process and append the instructions to the `statements` list.

Finally return the `statements` & `assert_cond_stat`.

Modifying Inputs

After taking the input we modify them into a format with which we can work with. We do this using `modifyInput()` function.

We identify whether an instruction in a program is read instruction or write instruction. Then we convert them into following format:

- For write instruction: [Program_no, write operation, variable, value]
- For read instruction: [Program_no, read operation, variable, variable we're reading into]

We also maintain a dictionary for all variables appearing in our programs as `all_variables_dict`. This will be useful when we check if the program satisfies the assert condition statement at the end of execution.

```
def modifyInput(statements):
    all_variables_dict = {}
    for i in range(len(statements)):
        statement = statements[i]
        for j in range(len(statement)):
            expression = statement[j]
            temp = expression.split("=")
            left = temp[0]
            right = temp[-1]

            if left not in all_variables_dict:
                all_variables_dict[left] = 0

            if(left == 'x' or left == 'y' or left == 'z'):
                statements[i][j] = str(i)+' ' + 'w' + ' '+left+' '+right
            else:
                statements[i][j] = str(i) +' '+ 'r' +' '+ right+' '+left

    return statements,all_variables_dict
```

Generating all possible executions

Using the `combine()` and `permStatements()` functions we generate all possible executions.

`combine(list1, list2):`

This method takes two lists as arguments each representing a list of instructions for a process. We then permute these two lists to get all possible concurrent executions with these 2 processes. We append each execution to the `results` list and return it to the `permStatements()` function.

```
def combine(lst1,lst2):
    comb_results = []
    for locations in itertools.combinations(range(len(lst1) + len(lst2)),
len(lst2)):
        result = lst1[:]
        for location, element in zip(locations, lst2):
            result.insert(location, element)
        comb_results.append(result)
    return comb_results
```

`permStatements(statements):`

In this function we pass the `statements` list which contains list of instructions corresponding to each process. If the number of processes is 1 then we simply return the statements list in perm_of_statements else we take 2 processes and create their permutation using `combine()` function. We store the result in a temporary variable.

Then we take all the remaining processes one-by-one and permute them with previous permutation. Thus, generating every possible concurrent execution with each process. We finally copy the result from temporary variable to `perm_of_statements` list and return it.

```
def permStatements(statements):
    if (len(statements) == 1):
        perm_of_statements = statements
    else:
        lst1 = statements[0]
        lst2 = statements[1]
        temp1 = combine(lst1, lst2)

        for i in range(2, len(statements)):
            temp2 = []
            for t in temp1:
                for x in combine(t, statements[i]):
                    temp2.append(x)

            temp1 = temp2.copy()

        perm_of_statements = temp1

    return perm_of_statements
```

Getting indices corresponding to read & write operation

We use the `get_index()` function to get indices corresponding to the read and write operations on a variable in a concurrent execution. Store the result in `indices` list which we return at the end of function execution. This list is later useful to get _rf _and ws *corresponds* to each concurrent execution.

```
def get_index(op,variable,perm_of_statement):
    indices = []
    for i in range(len(perm_of_statement)):
        temp = perm_of_statement[i].split(" ")
        if(temp[1]==op and temp[2]==variable):
            indices.append(i)
    return indices
```

Converting instructions to desired form

We use `convertExpressionToOrg()` function to get the instructions in desired format for *trace*, _rf _and *ws*.

```
def convertExpressionToOrg(expression):
    temp = expression.split(" ")
    process_no = temp[0]
    op = temp[1]
    variable = temp[2]
    value = temp[3]

    final_expression  = ""
    if(op == 'w'):
        final_expression = final_expression + variable+" = "+value
    if(op == 'r'):
        final_expression = final_expression + value + " = "+ variable

    return final_expression
```

rf Instructions

`getRF()` function is used to get all the read-from edges. For this we first get the list of write and read instructions for each variable using the `get_index()` function. Take the write instruction index latest to read the instruction index corresponding to a variable. We then convert the two instructions to readable format using `convertExpressionToOrg()` function. The result is appended to the `rf` list.

```
def getRF(perm_of_statement):

    rf = []
    for var in ['x','y','z']:
        w_var_indices  = get_index('w',var, perm_of_statement)
        r_var_indices = get_index('r', var, perm_of_statement)

        for r_var_index in r_var_indices:
            w_var_index = None
            for i in range(len(w_var_indices)-1,-1,-1):
                if(w_var_indices[i]<r_var_index):
                    w_var_index = w_var_indices[i]
                    break



            if(w_var_index != None):

rf.append([convertExpressionToOrg(perm_of_statement[w_var_index]),convertExpressio
nToOrg(perm_of_statement[r_var_index])])
            else:
                rf.append([var+' = '+'0',
convertExpressionToOrg(perm_of_statement[r_var_index])])

    return rf
```

ws Instructions

`getWS()` function works similar to `getRF()` function. It gets a list of indices corresponding to write operations on a variable. This list is used to get the write serialization order.

```
def getWS(perm_of_statement):
    ws = []

    for var in ['x','y','z']:
        w_var_indices = get_index('w',var, perm_of_statement)
        if(len(w_var_indices) <= 1):
            continue
        for i in range(len(w_var_indices)-1):

ws.append([convertExpressionToOrg(perm_of_statement[w_var_indices[i]]),convertExpr
essionToOrg(perm_of_statement[w_var_indices[i+1]])])

    return ws
```

Trace

We simply convert the instructions in the permutation to readable form to get the trace.

```
def getTrace(perm_of_statement):
    trace = []
    for expression in perm_of_statement:
        trace.append(convertExpressionToOrg(expression))
    return trace
```

Checking assert condition statement

We use the `checkAssert()` function to check the validity of a given assert statement. For this we take each expression in the concurrent execution and update the values in the all variable dictionary according to the operation. This way we are actually trying to execute this concurrent program and calculating the value of each variable. These values we later use to evaluate our assert condition through `flag = eval(assert_cond_stat)`. If the flag is set to `True` then we move on to the next execution, else we call the `assertViolateOutput()` in `main()`. `assertViolateOutput()` simply prints the *trace*, _rf _and _ws _which violate the assert statement.

```
def assertViolateOutput(trace,rf,ws):
    print("Error: Assertion Violation")
    print("Violating Trace: {}, rf relation: {}, co relation: {}".format(trace,
rf, ws))

def checkAssert(perm_of_statement,all_variables_dict,assert_cond_stat):

    for expression in perm_of_statement:
        temp = expression.split(" ")
        op = temp[1]
        variable = temp[2]
        value = temp[3]
        if(op == 'w'):
            all_variables_dict[variable] = int(value)
        if(op=='r'):
            all_variables_dict[value] = all_variables_dict[variable]

    variables = all_variables_dict.keys()
    for variable in variables:
        newvariable = "all_variables_dict['{}']".format(str(variable))
        assert_cond_stat = assert_cond_stat.replace(variable, newvariable)

    flag = eval(assert_cond_stat)

    return flag
```

Output

Once we are done with all the above operations and functions we collect the *trace*, _rf _and _ws _for each permutation. At the same time we make sure to get unique traces. This part takes place in main(). After we have collected all valid traces, rfs and wss we call output() which prints output to console in desired form.

```
def output(traces, rfs, wss):
    num_of_traces = len(traces)
    print("No. of traces = "+str(num_of_traces))
    for i in range(num_of_traces):
        print("{}-: Trace: {}, rf relation: {}, co relation:
{}".format(i+1,traces[i],rfs[i],wss[i]))
```

Parameter explanation for every function.

Function `assertViolateOutput`

> ```
> def assertViolateOutput(
>     trace,
>     rf,
>     ws
> )
> ```

This is to print Violating Trace.

param traces: This is a list of instructions in an order, which is considered as a possible trace.

param rfs: List of lists containing all the read from relations.

param wss: List of lists containing all write serializations.

Function `checkAssert`

```
def checkAssert(
    perm_of_statement,
    all_variables_dict,
    assert_cond_stat
)
```

param perm_of_statement: This is a list, having one possible interleaving of instructions of the programs.

param all_variables_dict: This is a dictionary having all the variables(global and local) and the value initialised to 0.

param assert_cond_stat: This is the string having assert condition only.

return:

flag: Boolean variable represent whether the assertion condition holds or not on the 'perm of statement'.

Function `combine`

```
def combine(
    lst1,
    lst2
)
```

param lst1: A list having instructions of a program in the same order as provided in input.

param lst2: Another list having instructions of a program in the same order as provided in input.

return:

comb_results:- This is a list of lists having all the inter-mixing possible of the instructions of programs which are in the two lists, keeping the order of instructions of both program maintained.

Function `convertExpressionToOrg`

```
def convertExpressionToOrg(
    expression
)
```

param expression: This is a string having instruction in modified form.

return:

final_expression:- Instruction in normal form as provided by the user.

Function `getRF`

```
def getRF(
    perm_of_statement
)
```

param perm_of_statement:This is a list, having one possible interleaving of instructions of the programs.

return:

rf:- This is a list of list containing all the read-from realtions between a write instruction and a read instruction on the same variable.

Function `getTrace`

```
def getTrace(
    perm_of_statement
)
```

param perm_of_statement: This is a list, having one possible interleaving of instructions of the programs.

return:

trace:- This is a list of instructions in an order, which is considered as a possible trace.

Function `getWS`

```
def getWS(
    perm_of_statement
)
```

param perm_of_statement: This is a list, having one possible interleaving of instructions of the programs.

return:

ws:- This is a list of list containing all the write serialization between two write instruction on the same variable.

Function `get_index`

```
def get_index(
    op,
    variable,
    perm_of_statement
)
```

param op: character referring to operation, 'w' means write and 'r' means read.

param variable: variable in consideration, such as 'x', 'y' , 'z'.

param perm_of_statement: This is a list, having one possible interleaving of instructions of the programs.

return:

indices(list):- indices of all the instruction which are performing the operation as provided in parameter "op" to the variable provided in the parameter "variable".

Function `inputProgram`

```
def inputProgram()
```

return:

statements:- This is a list of lists where each list has the instructions of a program, hence consist all the programs including the initialisation of the global variable as the default program.

assert_cond_stat:- This is the string having assert condition only.

Function `main`

```
def main()
```

Main function, execution starts from here.

Function `modifyInput`

```
def modifyInput(
    statements
)
```

param statements: This is a list of lists where each list has the instructions of a program.

return:

statements:- This is a list of lists where each list has the instructions of a program but the simple instruction is modified in a manner which is easy for the program to identify read and write operations.

all_variables_dict:- This is a dictionary having all the variables(global and local) and the value initialised to 0.

Function `output`

```
def output(
    traces,
    rfs,
    wss
)
```

This is to print final valid traces.

param traces: This is a list of instructions in an order, which is considered as a possible trace.

param rfs: List of lists containing all the read from relations.

param wss: List of lists containing all write serializations.

Function `permStatements`

```
def permStatements(
    statements
)
```

param statements: This is a list of lists where each list has the instructions of a program.

return:

perm_of_statements:- This is a list of list having all the inter-mixing of instructions of 'n' programs, keeping the order of instructions of a single program maintained.