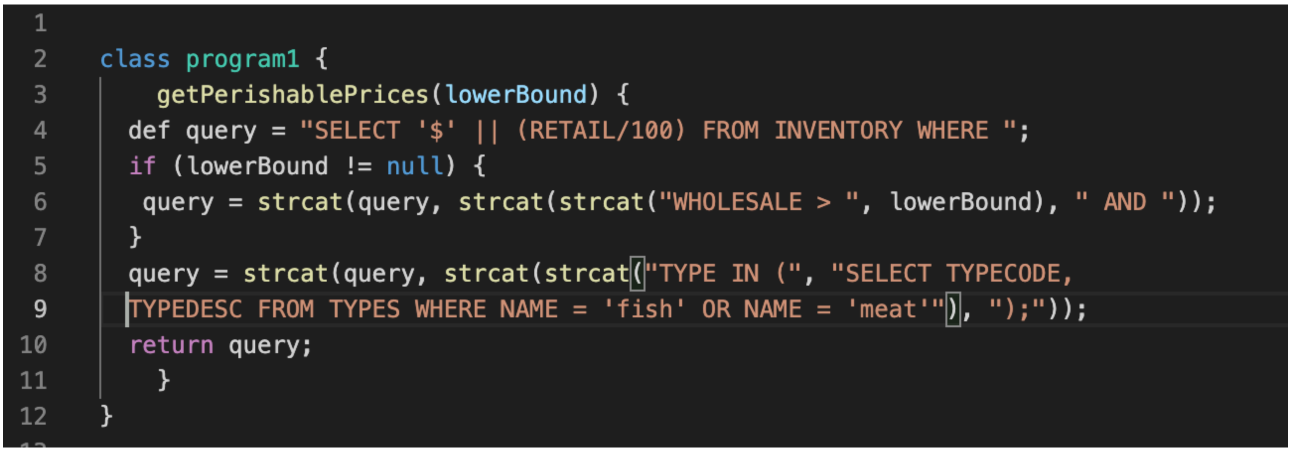
# Static Analysis of String Values in LiSA

Student Name: Zubair Ahmad (956603)
Teacher Name: Pietro Ferrara

## Introduction

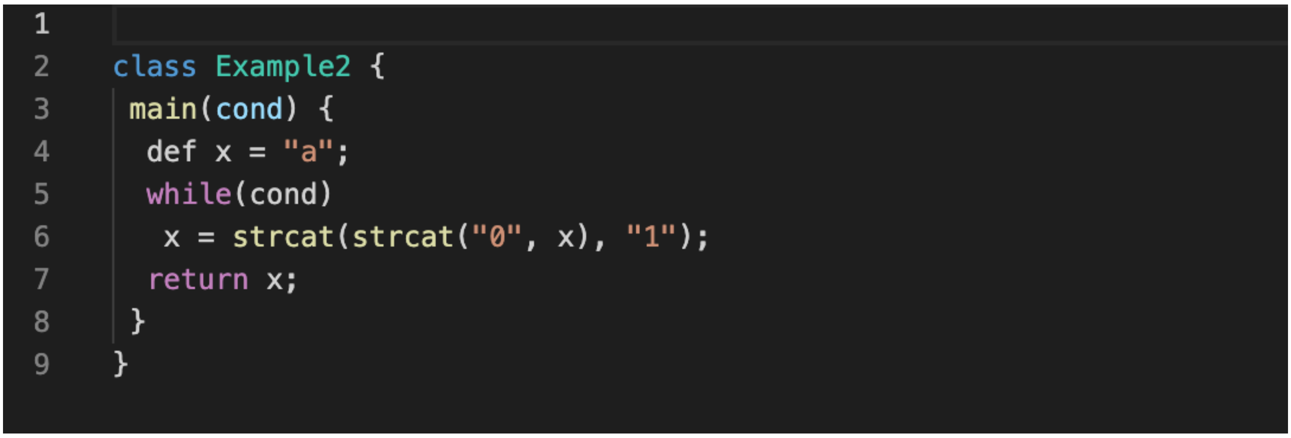
In this report, we discuss the application of static analysis of string values for safe and secure compilation. In particular we utilize ‘Library of Static Analysis (LiSA)’ a tool designed by ‘Software and System Verification (SSV)’ research group Ca’Foscari University Venice, Italy. LiSA is based on suite of abstract semantics which implements theory of abstract interpretation.

Throughout this report, we will utilize two examples as shown in figure (1) and (2). The first IMP program fig (1) concatenates various strings and dynamically builds an SQL query. Our main focus will be on SQL query resulting from string concatenation and checking out the execution of program and output. The second program in fig (2) produces strings of form $0^n\ a\ 1^n$ which modifies the string inside the loop.



```

class program1 {
    getPerishablePrices(lowerBound) {
  def query = "SELECT '$' || (RETAIL/100) FROM INVENTORY WHERE ";
  if (lowerBound != null) {
   query = strcat(query, strcat(strcat("WHOLESALE > ", lowerBound), " AND "));
  }
  query = strcat(query, strcat(strcat("TYPE IN (", "SELECT TYPECODE,
  TYPEDESC FROM TYPES WHERE NAME = 'fish' OR NAME = 'meat'"), ");"));
  return query;
    }
}
```

Figure (1)



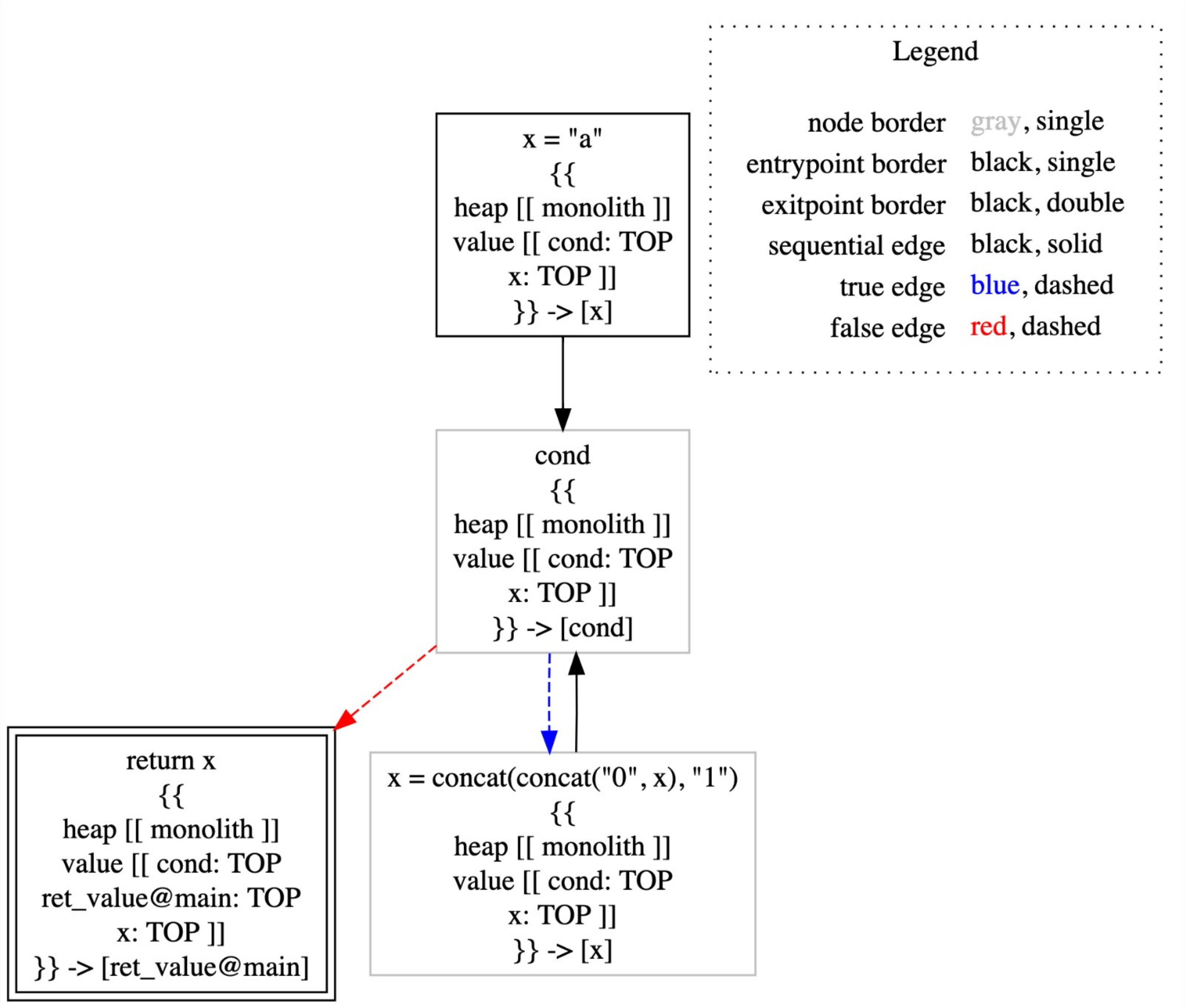
```

class Example2 {
 main(cond) {
  def x = "a";
  while(cond)
   x = strcat(strcat("0", x), "1");
  return x;
 }
}
```

Figure (2)

## CHARACTER INCLUSION

The results of the analysis of program using character Inclusion (CI) are depicted in Figure (1). At the beginning, the variable query is related to a state that contains the abstraction of line 4. The value of 'lowerbound' is unknown, so after the instruction 4 and 6, the lub between abstract values must be computed. After instruction 6, it will contain the all the character because they are all concatenated. Then, after the if statement, the abstract value of query contains the abstraction of line 4.  The variable query is related (line 8) to a state that contains the abstraction of instruction 8. At line 8 and 9, query is concatenated all the characters included. Then, at the end of the given code, query surely contains all the characters and it may contain any character, since we possibly concatenated in query an input string (lowerbound).

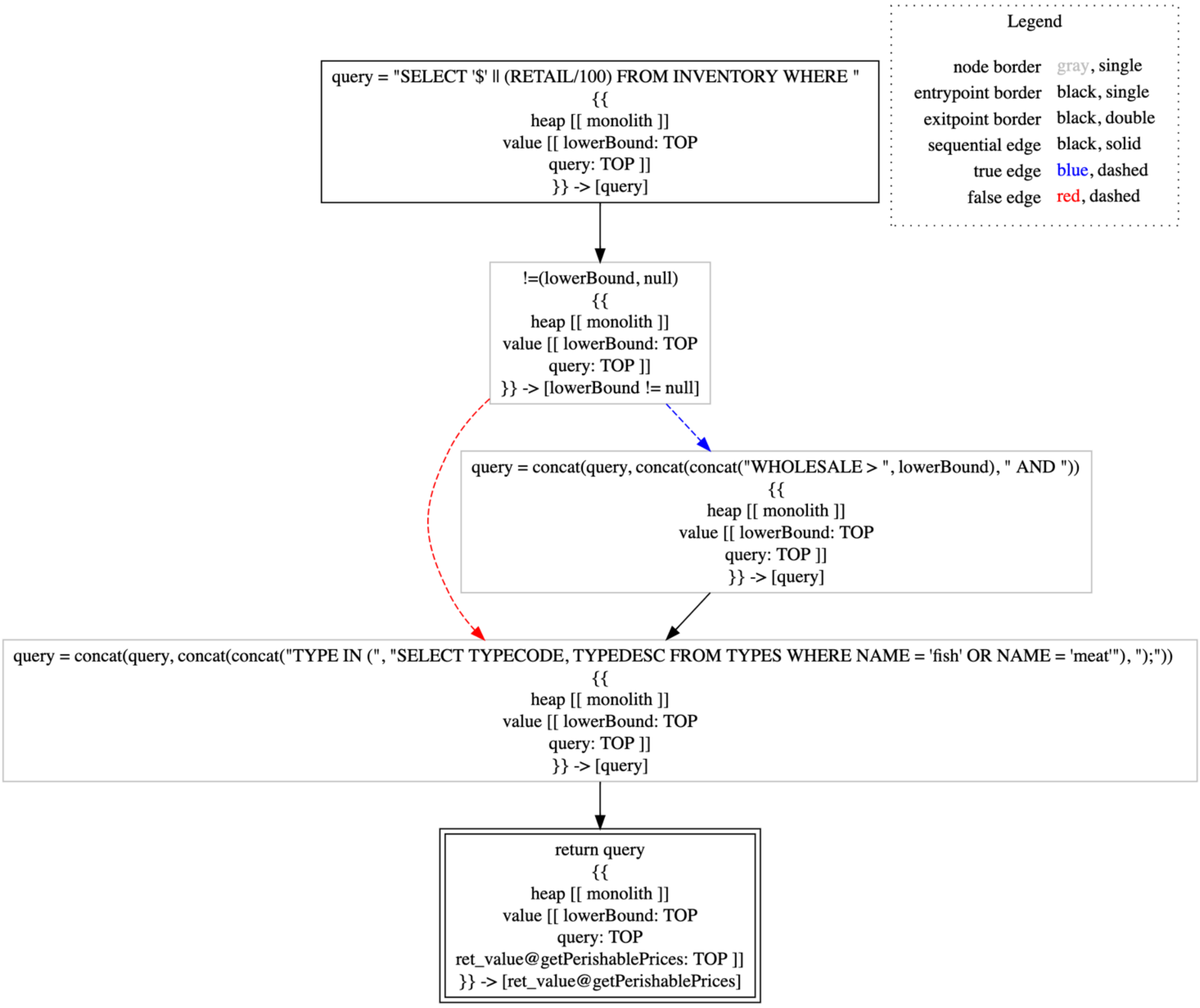
For program in Figure (2) we see in control flow graph that at the very start of the program after instruction 4,  x surely contains the character 'a'. After the first iteration of the loop (line 5), x surely contains 'a', '0' and '1'. At line 7 we report the least upper bound between the value of x before entering the loop (line 4) and the value after the loop (line 7): variable x surely contains the character 'a', and it also may contain the characters '0' and '1'. This is the final result of the program. Here the analysis converges immediately after the second iteration, since the abstract value obtained after two iterations (that is, ({0, a, 1}, {0, a, 1})) is the same as the one obtained after one iteration.


Legend
node border gray, single
entrypoint border black, single
exitpoint border black, double
sequential edge black, solid
true edge blue, dashed
false edge red, dashed
x = "a" {{ heap [[ monolith ]] value [[ cond: TOP x: TOP ]] }} -> [x]
cond {{ heap [[ monolith ]] value [[ cond: TOP x: TOP ]] }} -> [cond]
return x {{ heap [[ monolith ]] value [[ cond: TOP ret_value@main: TOP x: TOP ]] }} -> [ret_value@main]
x = concat(concat("0", x), "1") {{ heap [[ monolith ]] value [[ cond: TOP x: TOP ]] }} -> [x]

## PREFIX AND SUFFIX

For figure (1), at line 4, query contains the whole string as both prefix and suffix. ‘lowerbound’ is an input, so its prefix and suffix are both empty. After the concatenation at line 6, the prefix will be equal to line 4, the suffix to line 6 because we keep the prefix of the first string being concatenated and the suffix of the last one. Since the value of ‘lowerbound’ is unknown, we must compute the least upper bound between the abstract values of query after lines 4 and 6. The variable per is associated at line 8 for both the prefix and the suffix. At the end of the analysis, we get that the prefix of query is string in line 4 and its suffix is line 8

For figure (2), we know that the prefix and suffix of x are both an ‘a’ character before entering the loop. After the first iteration of the loop we get that the prefix of x is ‘0’ and its suffix is ‘1’. The least upper bound of such state with the state before the loop (prefix and suffix are both an ‘a’ character), unfortunately goes to TOP. Then, we reached convergence after just one iteration (since the least upper bound of any element with the TOP value returns always the TOP value), but we lost all the information


Legend
node border gray, single
entrypoint border black, single
exitpoint border black, double
sequential edge black, solid
true edge blue, dashed
false edge red, dashed

```
query = "SELECT '$' || (RETAIL/100) FROM INVENTORY WHERE "
{{
heap [[ monolith ]]
value [[ lowerBound: TOP
query: TOP ]]
}} -> [query]

!=(lowerBound, null)
{{
heap [[ monolith ]]
value [[ lowerBound: TOP
query: TOP ]]
}} -> [lowerBound != null]

query = concat(query, concat(concat("WHOLESALE > ", lowerBound), " AND "))
{{
heap [[ monolith ]]
value [[ lowerBound: TOP
query: TOP ]]
}} -> [query]

query = concat(query, concat(concat("TYPE IN (", "SELECT TYPECODE, TYPEDESC FROM TYPES WHERE NAME = 'fish' OR NAME = 'meat'"), ");"))
{{
heap [[ monolith ]]
value [[ lowerBound: TOP
query: TOP ]]
}} -> [query]

return query
{{
heap [[ monolith ]]
value [[ lowerBound: TOP
query: TOP
ret_value@getPerishablePrices: TOP ]]
}} -> [ret_value@getPerishablePrices]
```

**BRICKS**

For figure (1), at line 4 we represent query with a single brick with a singleton set (containing the string associated to query). The variable ‘lowerbound’ has an unknown value, so it is associated to TOP. At line 6 we concatenate the value of query to ‘lowerbound’, ‘WHOLESALE’ and ‘AND’ and we obtain a list of four bricks: the first two are made up by a singleton set (containing, respectively, line 4 and ‘WHOLESALE’ ) and the third one is TOP (because of ‘lowerbound’), and the fourth one is made up by a singleton set (containing ‘AND’). This means that, just after line 6, the string associated to query starts with line4 + ‘WHOLESALE’, then it has an unknown part, and then it ends with ‘AND’. Then, we have to compute the lub between the values of query after lines 4 and 6. At line 9, the abstract value of the variable is composed by a single brick with a singleton set (containing the string associated to query).

For figure (2), after line 1 the abstract value associated to x is a single brick with a singleton set (containing “a”). After the first iteration of the loop, the result of the concatenation is made up by three bricks, all of them with a singleton set (containing, respectively, “0”, “a” and “1”). To compute the least upper bound between the values after the first and second iterations (we do not know how many iterations the loop will do), We can see that, after each iteration, we obtain an abstract value which first and last bricks have an augmented range with respect to the value in the previous iteration: min is always zero, but max increases by one at each iteration. The convergence of the analysis is obtained through to the use of the widening operator. The result is almost optimal: the imprecision is due to the fact the number of occurrences of 0s and 1s are not restricted to be the same. For example, 0a11 is a concrete value represented by our resulting abstraction, but we know that this string can never be produced by the program.