Programming Assignment 2 WRITTEN QUESTIONS- CSCI-561 - Artificial Intelligence
SOLUTIONS

Chetan Ankola
1895488595
Last name: ankola

QUESTIONS AND ANSWERS:

1. The call to Occur-Check? in Unify-Var is to maintain soundness. Often, this call is omitted to speed up running time, albeit by risking unsound inferences. Provide an empirical study showing how much having an Occur-Check? call slows the running time.

1 You should:

• Provide non-trivial examples where there is no unifier found with the presence of Occur-Check? (and there is a unifier without it).

• Analyze and demonstrate empirically how the running time of your program changes with the presence/absence of Occur-Check?. Experiment with expressions of differing length, as well as structure (lists of lists, for example).

• Provide a brief qualitative analysis of your findings.

Note: This question is very open-ended, and worth a considerable portion of your grade. It is your duty to provide a self-contained analysis that demonstrates both the behavior of the algorithm, and your ability to run experiments and analyze results.

ANSWERS

Occur-Check is the considerably a time consuming step in the whole algorithm, since it involves search. Omission of Occur-Check() will easily increase search time of the overall algorithm depending on the size of the Compound expression being searched.

for eg: in Occur-Check(var, x) where var=a and x={ g(g(g(g(g(g(g(g(g(g(x)))))))))))))))))) }
In my Algorithm I perform a Linear search by splitting the Compound expression x into tokens which is a pure O(N) complexity if sizeof var is 1 else its of complexity O(N*size(var)) which is quite time consuming around O(N square) .

On the upside if we had ommitted the check, we would saved that amount of time.
However since we do not perform occur check, and if there is an occur_check failure then we end up checking all other values before we realize “NO SUBSTITUTION IS POSSIBLE”. If we had done an Occur_check and if we had caught the variable occurring inside the other matching expression , then we would recursively pass failure and reduce the number of calls to function.

for eg in this unifying example:

@LHS:
plus (L1 (x y L2 (a b c)) age)
@RHS:
plus (L3 (plus (plus (x) 2 L4 (1 2 3)) z)

With occur check

UNIFY([plus[[x y [a b c]] age]], [plus[[plus[plus[x]2 [2 3]]z]], {})
UNIFY([plus], [plus], {})
UNIFY([[[x y [a b c]] age]], [[[plus[plus[x]2 [2 3]]z]], {})
UNIFY([[x y [a b c]]], [[plus[plus[x]2 [2 3]]z]], {})
UNIFY([x], [plus[plus[x]2 [2 3]]], {})
UNIFY_VAR:([x] ,[plus(plus(x)2L4(123))] , {})
OCCUR_CHECK FAILURE!!!!!! x<< occurs is plus (plus (x) 2 L4 (1 2 3))
UNIFY([[y [a b c]]], [[z]], {$#})
UNIFY([[age]], [[]], {$#})
LHS:plus (L1 (x y L2 (a b c)) age)
RHS:plus (L3 (plus (plus (x) 2 L4 (1 2 3)) z)
NO SUBSTITUTION POSSIBLE

Without occurcheck
UNIFY([plus[[x y [a b c]] age]], [plus[[plus[plus[x]2 [2 3]]z]], {})
UNIFY([plus], [plus], {})
UNIFY([[[x y [a b c]] age]], [[[plus[plus[x]2 [2 3]]z]], {})
UNIFY([[x y [a b c]]], [[plus[plus[x]2 [2 3]]z]], {})
UNIFY([x], [plus[plus[x]2 [2 3]]], {})
UNIFY_VAR:([x] ,[plus(plus(x)2L4(123))] , {})
UNIFY([[y [a b c]]], [[z]], {x:plus(plus(x)2L4(123))})
UNIFY([y], [z], {x:plus(plus(x)2L4(123))})
UNIFY_VAR:([y] ,[z] , {x:plus(plus(x)2L4(123))})
UNIFY([[[a b c]]], [[]], {x:plus(plus(x)2L4(123)) y:z})
UNIFY([[age]], [[]], {$#})
LHS:plus (L1 (x y L2 (a b c)) age)
RHS:plus (L3 (plus (plus (x) 2 L4 (1 2 3)) z)

We would see that there are 9 calls to UNIFY() in case of no call to occur check and just 7 calls in case of making occur check calls plus O(Nsquare) string matching for each occur_check call.

In the above example both with and without occur-check are able to find the result to be no-substitution since my code takes care of it. however in most cases, lack of occur-check can result in an infinite substitution i.e cyclic substitution since the variable occurs in the expression

to be unified with.

for eg:
plus (plus (x))
@RHS:
plus (plus (L1 (L1 (L1 (x)))))

with occur check the output is “NO SUBSTITUTION”
and without occurcheck the output is : Substitution:{x:L1(L1(L1(x))) }
which on being substituted will result in infinite loop of substitution
for eg: if x was being used in some other expression it would make no meaning to have x within its own substitution.

So to conclude:
Although occur check comes at some search cost, its necessary to have an occur check if we want to prevent getting wrong results or prevent system crashes in case of infinite substitutions in a real time system. In fact we can reduce the “Search” time complexity by
creating hash of tokens of the expression x, and then making a hash search of the var instead of linear search which will which will bring down search time to O(sizeof(var))

2. In this assignment, we ask you to find the most general unifier (MGU). Suppose instead we asked forsimply any unifier as fast as possible.
a How would you change your program to so as to find a unifier (albeit it not the most general one)faster? You don’t actually need to change your program to add this, just describe what you would do.
b Why, in general, is a more general unifier preferred over a less general one?
SOLUTION 2:
2a.

A simple change to the program would be to change the UNIFY_VAR() function, where we assign a value or expression to a var. Instead of updating the substitution string with the var/ (value or expression) ; we can check if any other variable exists within the expression, which hasnt been assigned a value and run a UNIFY for that variable, till we find substitution for that and assign it to the previous expression.

2b.MGU is different from other unifiers in a way that it finds substitution which might be in terms of other variables/substitutions. i.e keep 2 expression side by side and check the first sub-expression they mismatch. If they don mismatch we perform an assignnment without checking if the unifiers within have a substitution or not.
for eg: LHS: plus(x), y , x and RHS: z, 5 , 5
our MGU would look like z/plus(x) , y/5 x/5
instead of z=plus(5) y/5 x/5

If we see this we would realize that MGU would be preferred for the following reasons:
1>Since in MGU every variable is assigned values sometimes in terms of other variables or a combination, the space required would be less. for eg: in the Database we could just store the index instead of the whole computed value.
2> MGU is faster since we dont go ahead recursively trying to find the exact value of the variable. We stop once we find that an expression is a match for a variable without worrying what the expression contains (offcourse we do the OCCUR_CHECK to make sure that variable itself doesnt occur in the expression)

3. How did you test your code?
Do not simply enumerate test cases, but describe how you went about testing for specific algorithmic issues.
ANSWER3:
1. I created a Test suite, where with a particular flag enabled say -t1 we run test case 1 i.e a particular case gets executed.

2. Individual test cases were created to test UNIFY_VAR, UNIFY, OCCUR_CHECK etc where the functions were tested directly with a intermediate test inputs.

3.I used “DEBUG1” flag in my code to enable trace and prints which are more useful to me and enumerates all cout statements, for eg: the contents of the file, the tokens generated,

4. I wrote universal functions like getargs, gettokens etc which would fetch the first and rest of the arguments or split a string into tokens based on space delimiter and return var/val pairs. Since these functions were tested thoroughly, and since the functions were consistently used the amount of debugging required was very less.

5. I did not use STL map for substitution stringbut used map to find that if a token is a variable or constant etc. Since I did not use map, I wrote a custom parser to parse the tokens of the substitution string and had to make sure that there is no memory leak or pointer exceptions.

6. As far as testing the correctness of the algorithm is concerned I have tested it with a wide range of test expressions including empty strings, and no where my program has shown segmentation fault.

7. Apart from all these I have written test cases to check when the substitution string gets updated properly or not, and if var belongs to substitution string already etc.