

CSE560: Homework 8

SITUATION SEMANTICS PLANNER

Make a forward planner that works with Situation Semantics. Your forward planner will need a depth-first theorem prover to evaluate the 'poss'. You could use a variation of your theorem prover from earlier homeworks, but that (a) supports explicit unification, (b) supports the 'not' as negation as failure, but without the delaying mechanism, and (c) that searches for all solutions for a query, rather than just the first one.

You can do this assignment in Prolog or in Tcl. For those of you doing this in Tcl, you can use the theorem prover below. The first argument is the query. The second argument can be set to "", and is only used if '::ProveDebug' is set to 1. It simply is a string that is used to indent each output line and can be used when doing negation as failure to show that you are doing an embedded proof.

```
source hw4standard.tcl

set ProveDebug 0

proc proveall {query indentStr} {
    set vars [findvariables $query {}]
    if {$::ProveDebug} {
        puts [format "%sVars in query are %s" $indentStr $vars]
    }
    set yes [linsert $vars 0 yes]
    set answer [linsert $query 0 $yes]
    if {$::ProveDebug} {
        puts [format "%sInitial answer clause is %s" $indentStr \
                    [prettyclause $answer]]
    }

    # the initial frontier is a list whose only element is the initial
    # answer clause
    set frontier [list $answer]
    set answers {}

    while {$frontier != {}} {
        # lets proving the top element in the frontier
        set answer [lindex $frontier 0]
        set frontier [lrange $frontier 1 end]

        if {[llength $answer] == 1} {
            set yesatom [lindex $answer 0]
            set answer {}
            if {$::ProveDebug} {
                puts [format "%sPROVED %s" $indentStr [prettyexpr $yesatom]]
            }
            foreach var $vars val [lrange $yesatom 1 end] {
                lappend answer $var $val
            }
            lappend answers $answer
            continue
        }
    }

    # give it fresh variables
    if {$::ProveDebug} {
```

```

    puts $indentStr
    puts [format "%sTrying to prove      : %s" $indentStr \
              [prettyclause $answer]]
}
set answer [freshvariables $answer]
set firstatom [lindex $answer 1]

set neighbors {}
set firstpred [lindex $firstatom 0]
if {$firstpred == "not"} {
    set result [proveall [lrange $firstatom 1 end] "| $indentStr"]
    if {$result == {}} {
        set neighbor [lreplace $answer 1 1]
        if {$::ProveDebug} {puts "$indentStr Neighbor : $neighbor"}
        set neighbors [list $neighbor]
    }
} elseif {$firstpred == "="} {
    set result [unify [lindex $firstatom 1] [lindex $firstatom 2] {}]
    if {$result != 0} {
        set neighbor [substitute [lreplace $answer 1 1] $result]
        if {$::ProveDebug} {puts "$indentStr Neighbor : $neighbor"}
        set neighbors [list $neighbor]
    } else {
        if {$::ProveDebug} {puts "$indentStr Unification failed"}
    }
} else {
    foreach r $::kb {
        # check if first atom in body of answer clause matches head of rule
        set subs [unify $firstatom [lindex $r 0] {}]
        if {$subs == 0} continue

        if {$::ProveDebug} {
            puts "$indentStr Using rule: [prettyclause $r]"
            puts "$indentStr      subs: $subs"
        }

        #create new answer clause (with correct substitutions)
        set newr [substitute $r $subs]
        set answercopy [substitute $answer $subs]
        set answercopy [concat [list [lindex $answercopy 0]] \
                              [lrange $newr 1 end] \
                              [lrange $answercopy 2 end]]
        lappend neighbors $answercopy
        if {$::ProveDebug} {
            puts "$indentStr Neighbor : [prettyclause $answercopy]"
        }
    }
}
set frontier [concat $neighbors $frontier]
}

if {$answers == {} && $::ProveDebug} {
    puts [format "%sNo solutions" $indentStr]
}
return $answers
}

```

I am also giving you a definition of **poss** and the primitive predicate symbols. I am also giving you what is true in the initial world. I am also giving you what the goal is. These are the answers from homework 7. I will send this to you after you have turned in homework 7.

```
set kb {\
    ...
    {{goal S} {on a t S} {on b c S}} \
}
```

Question 1

Before you attempt to write the forward planner, make sure you understand how to use the **proveall** procedure. Give the tcl code to call **proveall** to prove the following.

a is on **b** in the initial world

The robot is holding **a** after it picks up **a** off of **b** from the initial world.

The robot's hand is empty after putting **a** on the table, after picking up **a** off of **b** from the initial world.

The action of picking up **a** off of **b** is possible in the initial world.

Question 2

The previous question did not make use of **proveall**'s ability to return multiple answers. Let's do that now. Give the tcl code to do the following, and report what **proveall** returns as the answer.

Find all blocks that are on the table in the initial world.

Find all actions that are possible in the initial world.

Find all actions that are possible after we have picked up **a** off of **b** from the initial world.

Find all actions that are possible after we have put **a** on the table after we picked up **a** off of **b** from the initial world.

Question 3: BONUS

You need to write a forward planner that explicitly keeps a frontier of worlds. The frontier will begin with just the single element **init**. On each iteration (hint: while loop), you first see if you can prove that goal is true of the current world. If it is, then you can stop. Otherwise, you need to find all actions that are possible (using **proveall**) in the current world. For each possible action, you can construct the new world as **do(Action,World)**. This will give you all of the neighbors of the current world. You then add the neighbors to the frontier using a breath-first search strategy or iterative deepening.

Hand in a copy of your code and a copy of its output, that shows what it is doing. Condense your output so that it is a reasonable length.

Question 4: BONUS

Explain why we are not using a theorem prover that delays the **not** if it has variables in it.

Question 5: BONUS

In the definition of the primitive relations, one must refer to the action that was executed to bring you from the previous state to the current state. In the textbook and in the slides, the definitions ensured that the

action was possible in the previous state. Why is this not necessary in how you wrote your planner?

Question 6

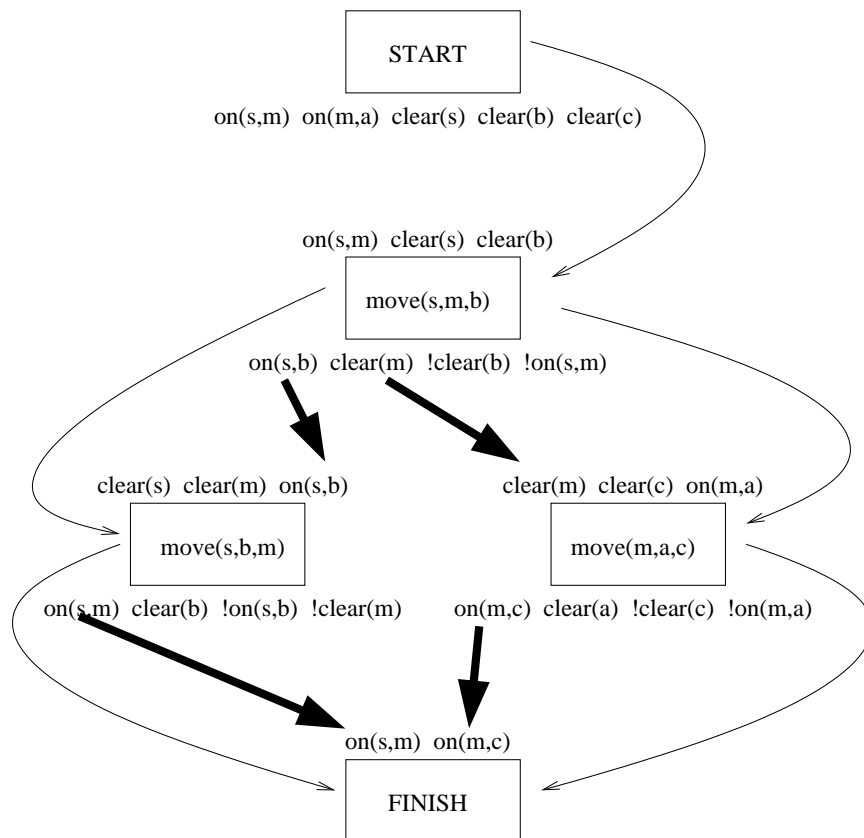
Part a: The Strips forward-planner that you wrote for homework 7 did not have the ability to use derived relations. Why was this?

Part b: Describe how you would have to change your Strips forward-planner to allow this.

Part c: Does the situation-semantics planner that you wrote for this question allow derived relations? Why or why not?

Part d: What primitive relations could you have instead defined as derived? Give their definition. (If you test out your definitions, make sure you get rid of the corresponding primitive relations and their facts for the initial world.)

POP PLANNER



In the partial plan above, thin lines show ordering constraints and thick lines show causal links.

Question 7

For the partial plan above, list all threat(s) (clobberers), and explain why each is a threat, and explain how each can be resolved.

Question 8

For the preconditions that have not yet been satisfied, they can be satisfied using the existing actions, and without introducing more threats. Draw causal links for them.

Question 9: BONUS

Implement a partial-order planner in Tcl.

Question 10

Exercise 9.1 from the textbook.