Universität des Saarlandes
Fachrichtung 6.2 – Informatik
J. Siekmann, S. Autexier, C. Benzmüller, C.E. Brown
C. Hahn

8th Theoretical Assignment in Artificial Intelligence (SS 2005)
Solutions

Exercise 8.1: **10 P**

Use unification grammars to represent the sentence *John loves Mary* in a feature sentence.

Solution:

$$
\begin{bmatrix}
x1 & \begin{bmatrix}
\text{category:} & \text{NP} \\
\text{head:} & \begin{bmatrix}
\text{det:} & \\
\text{root:} & \text{John} \\
\text{aggreement:} & \text{singular}
\end{bmatrix}
\end{bmatrix} \\
x2 & \begin{bmatrix}
\text{category:} & \text{VP} \\
\text{head:} & \begin{bmatrix}
\text{root} & \text{: love} \\
\text{tense} & \text{: present} \\
\text{agreement:} & \text{singular} \\
\text{object:} & \begin{bmatrix}
\text{category:} & \text{NP} \\
\text{head:} & \begin{bmatrix}
\text{root:} & \text{Mary} \\
\text{agreement:} & \text{singular}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
$$

Exercise 8.2: **20 P**

In the following example, we will show how the planning algorithm derives a solution to a problem that involves putting on a pair of shoes. In this problem scenario, Pat is walking around his house in his bare feet. He wants to put some shoes on to go outside. The actions are described in the following table. The following states are given `OnRightSock`, `OnLeftSock`, `OnBarefootLeft`, `OnBarefootRight`, `OnRightShoe` and `OnLeftShoe`.

	RightShoe	RightSock	LeftShoe	LeftSock
preconditions	OnRightSock	OnBarefootRight	OnLeftSock	OnBarefootLeft

- List all operators (inclusive *adds* and *deletes*, i.e. use `STRIPS` without negation)). (5 P)

Solution:

$Op($ACTION:$LeftSock,$
PRECOND:$OnBarefootLeft$
ADDS:$OnLeftSock$
DELETES:$OnBarefootLeft)$

$Op($ACTION:$RightSock,$
PRECOND:$OnBarefootRight$
ADDS:$OnRightSock$
DELETES:$OnBarefootRight)$

$Op($ACTION:$LeftShoe,$
PRECOND:$OnLeftSock$
ADDS:$OnLeftShoe$
DELETES:$OnLeftSock)$

$Op($ACTION:$RightShoe,$
PRECOND:$OnRightSock$
ADDS:$OnRightShoe$
DELETES:$OnRightSock)$

- Introduce two additional operators for putting a hat and a coat on, with the preconditions that the hat can only be put on if both pairs of shoes are on. There a no preconditions for putting on the coat. Add the adds and deletes of this two operators. (5 P)

Solution:

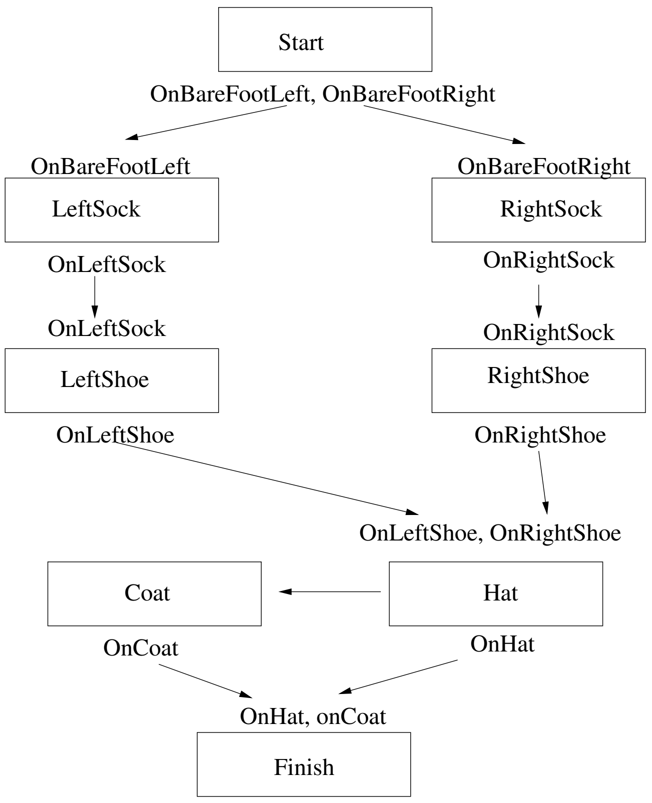
$Op($ACTION:$Hat,$
PRECOND:$OnLeftShoe \wedge OnRightShoe$
ADDS:$OnHat$
DELETES: -)

$Op($ACTION:$Coat,$
PRECOND: -
ADDS:$OnCoat$
DELETES: -)

- Give a partial-order plan that solves this shoe problem. How many feasible plans do exist? (10 P)

Solution:

Partial order plans can be executed in many possible ways. It is possible for Pat to place either of his socks (left or right) first. From there, he can place the other sock or a boot on. If he chooses to place both socks on first, he then can choose between boots. If he placed the boot on, he would have to put his other sock on, etc. Overall there exists 36 different plans.



Note: It was not entirely clear that we would have to produce a plan for putting on the the hat and the coat. If you generate a plan for putting on the shoes only, then the plan stops where the Hat operator is. In that case you only have 6 possible linearisations.

Exercise 8.3: **20 P**

Develop a functional version of the STRIPS algorithm presented in the lecture. This version should return a feasible plan if the problem is solvable. Use pseudocode (similar to the lecture slides) to present your algorithm and explain additionally your algorithm using natural language.

Solution:

We use a Lisp-like notation for a functional version of the STRIPS procedure. The arguments are:

- *S : the current state (initially the initial state)*
- *P : the plan computed so far to reach that state S (initially empty)*
- *G : the set of goals to be fulfilled starting from state S*
- *O : the list of operators*

Here is an example algorithm. The initial call is STRIPS(I, NIL, G, O).

```
(defun STRIPS (S, P, G, O)
 (if (= ($G \setminus S$) nil) (values T S P)
  (some '(lambda (g)
           (some '(lambda (o)
                    (let (($\delta$ (some-effect-unifies-with o g)))
                       (if (= $\delta$ 'fail)
                           nil ;; we backtrack over the operator or goal
                         ;; otherwise ...
                         (multiple-value-bind (Success? S' P')
                              (STRIPS S' P' $\delta$(Preconditions(o)), O)
                           (if Success?
                               (let ((o' (applicable-ground-instance-of-operator
                                                  S'  $\delta$(Preconditions(o)))))
                                   ;; recursion
                                   (STRIPS (result S' o') (addlast P' o') G O))
                            ;; else, if preconditions could not be established
                            nil ;; backtrack over operator or goal)))))
                  O))
          $G \setminus S$)))
```

This relies on the following functions, which we assume to be given

- `some-effect-unifies-with o g`*: Which computes the substitution to be applied on* `o` *such that one of its effects unifies with* `g`*. If there is no such effect, it returns* `'fail`
- `applicable-ground-instance-of-operator S o`*: Computes a ground instance of* `o` *which is applicable in* `S`*. By looking at the context where this function is called, it can never fail.*

The point of this exercise was that the students find out how to extract the plan compute by the strips algorithm as well as make the structure of the search algorithm more explicit than it is in the original procedural notation with side effects.

So there may be different formulations and notations, but the important things are

- *The main points of recursion are correctly identified and the correct arguments are passed.*
- *The plan is assembled by adding the ground instance of the operator after its preconditions have been established.*
- *The state the planner is in (S) is correctly maintained*
- *The backtracking is correctly done (we handled backtracking using the semantics of the lisp* `some`*). But there may be other ways.*

Exercise 8.4: **20 P**

Assume, Pat got drunk last night and has to go the supermarket to buy groceries. Unfortunately, Pat got involved in an onion eating contest and a mud-wrestling match last night. Pat is currently sleeping and is hungry. One last thing that can be said about Pat is that he is lazy and does not like to do anything more than he has to. How should Pat get ready to go to the supermarket?

- Define the problem by specifying the environment and operators. The environment should comprehend the states `HairMessy`, `Dressed`, `Clean`, `Hungry`, `Sleeping`

Solution:

$Op($ACTION:$WakeUp$,
  PRECOND:$Sleeping$
  EFFECT:$\neg Sleeping)$

$Op($ACTION:Eat,
  PRECOND:$Hungry \wedge \neg Sleeping$
  EFFECT:$\neg Hungry)$

$Op($ACTION:$TakeShower$,
  PRECOND:$\neg Clean \wedge \neg Sleeping$
  EFFECT:$Clean)$

$Op($ACTION:$WashHair$,
  PRECOND:$HairMessy \wedge Clean \wedge \neg Sleeping$
  EFFECT:$\neg HairMessy)$

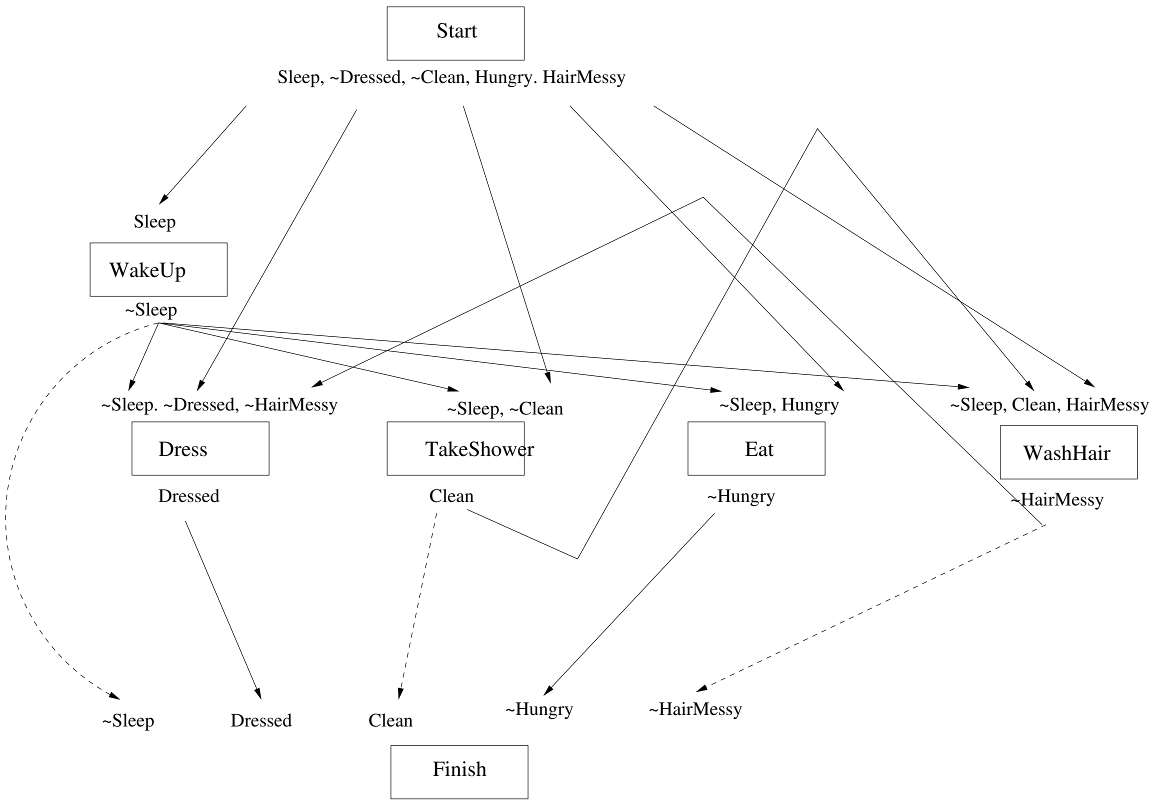
$Op($ACTION:$Dress$,
  PRECOND:$\neg Dressed \wedge \neg Sleeping \wedge \neg HairMessy$
  EFFECT:$Dressed)$

- Create a Minimal Partial Order Plan.

Solution:

Exercise 8.5: **30 P**

Consider a blockworld in which there is a *table*, and the blocks A, B, and C. Moreover, there are predicates $Clear(x)$ where x is a block and $On(x, y)$, where x is a block and y is a



block or the table. Furthermore, consider the following operators:

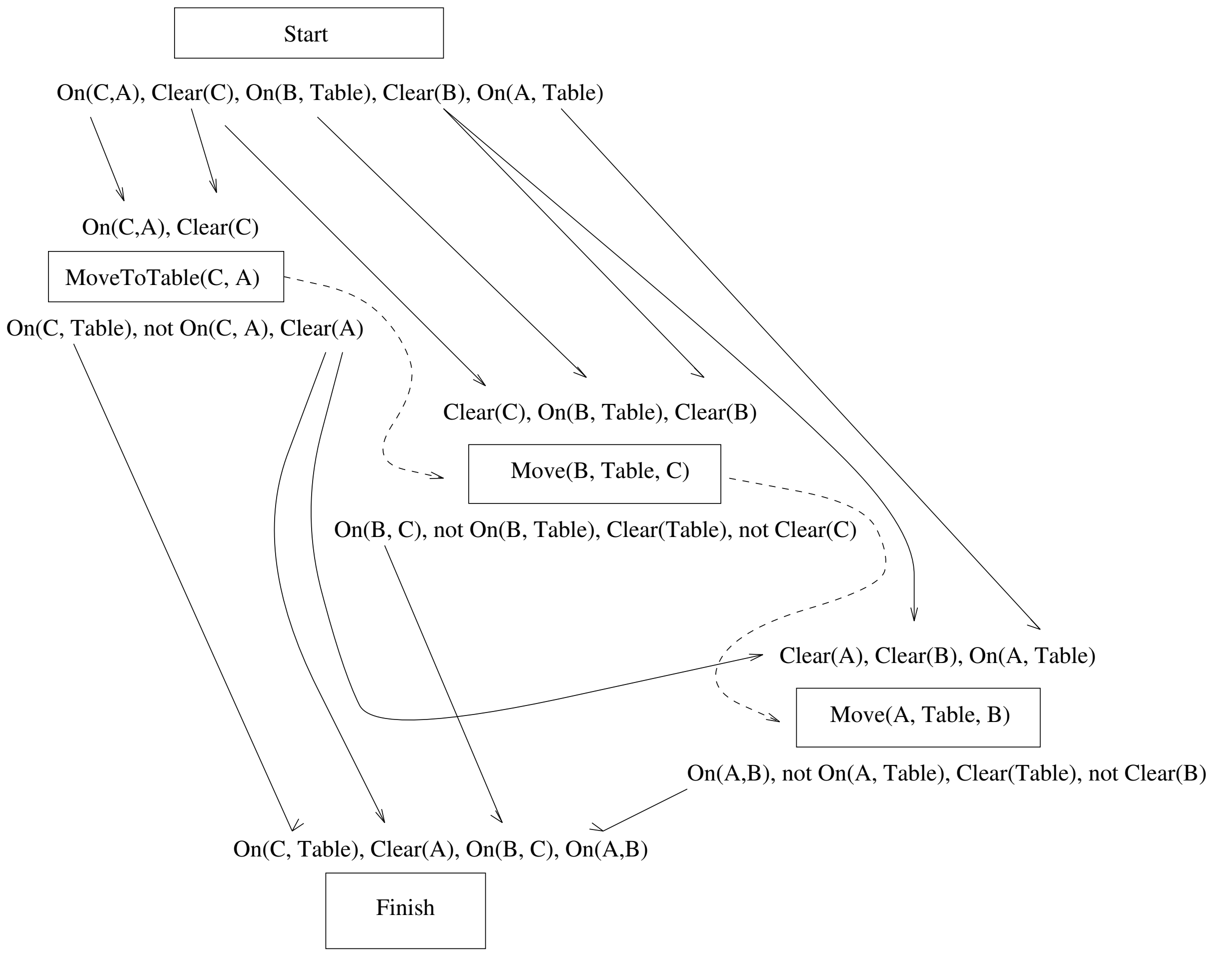
$Op($ACTION:$Start,$
  EFFECT:$On(C,A) \wedge On(A,Table) \wedge Clear(C) \wedge On(B,Table) \wedge Clear(B))$

$Op($ACTION:$Finish,$
  PRECOND:$On(A,B) \wedge On(B,C) \wedge On(C,Table) \wedge Clear(A))$

$Op($ACTION:$Move(b,x,y),$
  PRECOND:$On(b,x) \wedge Clear(b) \wedge Clear(y)$
  EFFECT:$On(b,y) \wedge Clear(x) \wedge \neg On(b,x) \wedge \neg Clear(y))$

$Op($ACTION:$MoveToTable(b,x),$
  PRECOND:$On(b,x) \wedge Clear(b)$
  EFFECT:$On(b,Table) \wedge Clear(x) \wedge \neg On(b,x))$

Explain in detail how a *partial order planner (POP)* generates a plan for the Sussman anomaly. Give the different steps during the planning process as well as a diagram that shows the final plan with causal links including preconditions and ordering constraints.


Start
On(C,A), Clear(C), On(B, Table), Clear(B), On(A, Table)
On(C,A), Clear(C)
MoveToTable(C, A)
On(C, Table), not On(C, A), Clear(A)
Clear(C), On(B, Table), Clear(B)
Move(B, Table, C)
On(B, C), not On(B, Table), Clear(Table), not Clear(C)
Clear(A), Clear(B), On(A, Table)
Move(A, Table, B)
On(A,B), not On(A, Table), Clear(Table), not Clear(B)
On(C, Table), Clear(A), On(B, C), On(A,B)
Finish

Solution:

S_{need}	c	S_{add}	*Bindings*
Finish	*On(A, B)*	*Move(x1,x2,x3)*	$x1/A, x3/B$
Move(A,x2,B)	*On(A,x2)*	*Start*	$x2/Table$
Move(A, Table, B)	*Clear(B)*	*Start*	
Finish	*On(C, Table)*	*MoveToTable(y1,y2)*	$y1/C$
MoveToTable(C, y2)	*On(C, y2)*	*Start*	$y2/A$
MoveToTable(C,A)	*Clear(C)*	*Start*	
Finish	*Clear(A)*	*MoveToTable(C,A)*	
Move(A, Table, B)	*Clear(A)*	*MoveToTable(C,A)*	
Finish	*On(B,C)*	*Move(z1,z2,z3)*	$z1/B, z3/C$
Move(B, z2, C)	*On(B, z2)*	*Start*	$z2/Table$
Move(B, Table, C)	*Clear(B)*	*Start*	
Move(B, Table, C)	*Clear(C)*	*Start*	

Threats: The `not Clear(C)` *effect of the action* `Move(B, Table, C)` *threatens the effect* `Clear(C)` *of the start action in* `MoveToTable(C,A)`*. To solve this problem,* `Move(B, Table, C)` *is shifted behind* `MoveToTable(C,A)`*. The effect of* `not Clear(B)` *of* `Move(A, Table, B)` *threatens the effect of* `Start` *which is used in the action* `Move(B, Table, C)`*. Therefore,* `Move(A, Table, B)` *is shifted behind* `Move(B, Table, C)`*.*