Cut, Not, and Fail

York University CSE 3401
Vida Movahedi

Overview

- Multiple solutions
- Cut
  - Examples
  - 3 reasons to use
- Not
- Fail
- Problems with using Cut

[ref.: Clocksin- Chap. 4]
[also Prof. Gunnar Gotshalks' slides]

Multiple Solutions

- Given a set of facts, e.g.
  father(mary, george).
  father(john, george).
  father(sue, harry).
  father(george, edward).

- A query such as :- father(X,Y). can generate multiple solutions (if user prompts with a semicolon):
  X= mary, Y= george;
  X= john, Y= george;
  X= sue, Y=harry;
  X= george, Y= edward

- And a query such as :- father(_,X). generates:
  X= george;
  X= george;
  X= harry;
  X= edward

Multiple Solutions (cont.)

- Example:

```
is_integer(0).
is_integer(X):- is_integer(Y), X is Y+1.
```

  The query :- is_integer(X). will get infinite number of integers:
  X=0;
  X=1;
  X=2; .... → **backtracking will go on forever!**

- Example:

  :-member(a, [a,b,a,a,b]).
  true;
  true;
  true;
  false. → **Only need to succeed once, waste of time!**

- In some cases, we need to have control over backtracking!

Library Example

- Allow access to basic facilities to clients with overdue books, otherwise allow access to general facilities:

```
facility(Pers, Fac):- book_overdue(Pers, Book), basic_facility(Fac).
facility(Pers, Fac):- general_facility(Fac).
client('A. Jones').
book_overdue('A. Jones', book100).
book_overdue('A. Jones', book200).
basic_facility(enquiries).
general_facility(borrowing).
```

- Go through all clients and find the facilities open to them:

```
:- client(X), facility(X,Y).
```

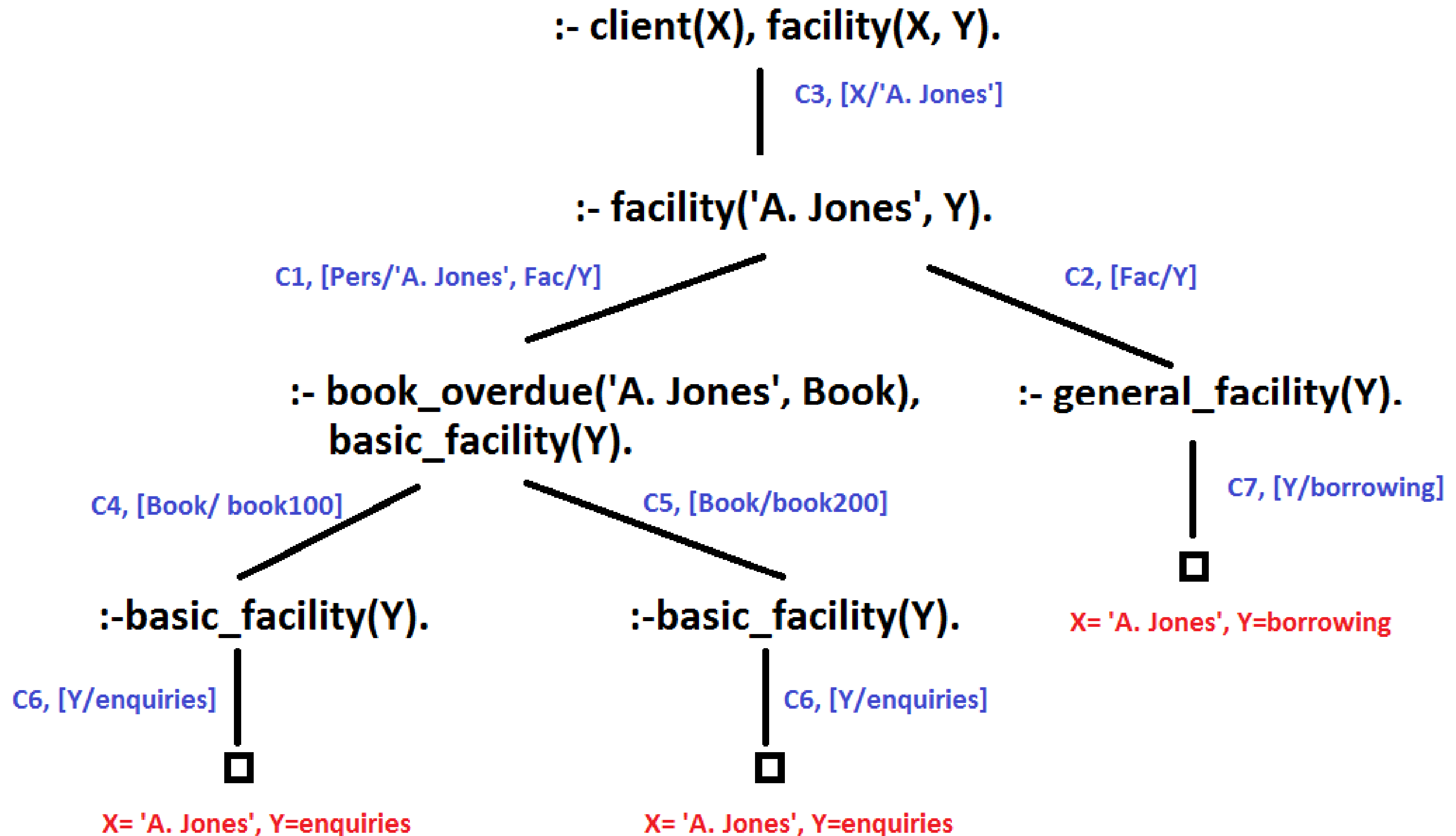
Library Example (cont.)

- The query will be answered:

  X= ‘A. Jones’, Y= enquiries;
  X= ‘A. Jones’, Y= enquiries;
  since A. Jones had two overdue books
  X= ‘A. Jones’, Y= borrowing
  certainly not what we want!
  Resolving with the second rule!

- Another example for the need to control backtracking!

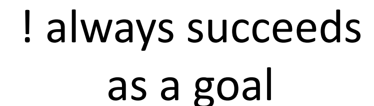
Search tree of Library Example



CUT !

- Changing the code as follows solves this problem:

```
facility(Pers, Fac):- book_overdue(Pers, Book), !, basic_facility(Fac).
facility(Pers, Fac):- general_facility(Fac).
client('A. Jones').
book_overdue('A. Jones', book100).
book_overdue('A. Jones', book200).
basic_facility(enquiries).
general_facility(borrowing).
```



- Cut tells Prolog:
  - If you got this far, don't try resolving with the second rule for *facility*.
  - If you got this far, commit to values you have, for example for *Book*

What does ! mean?

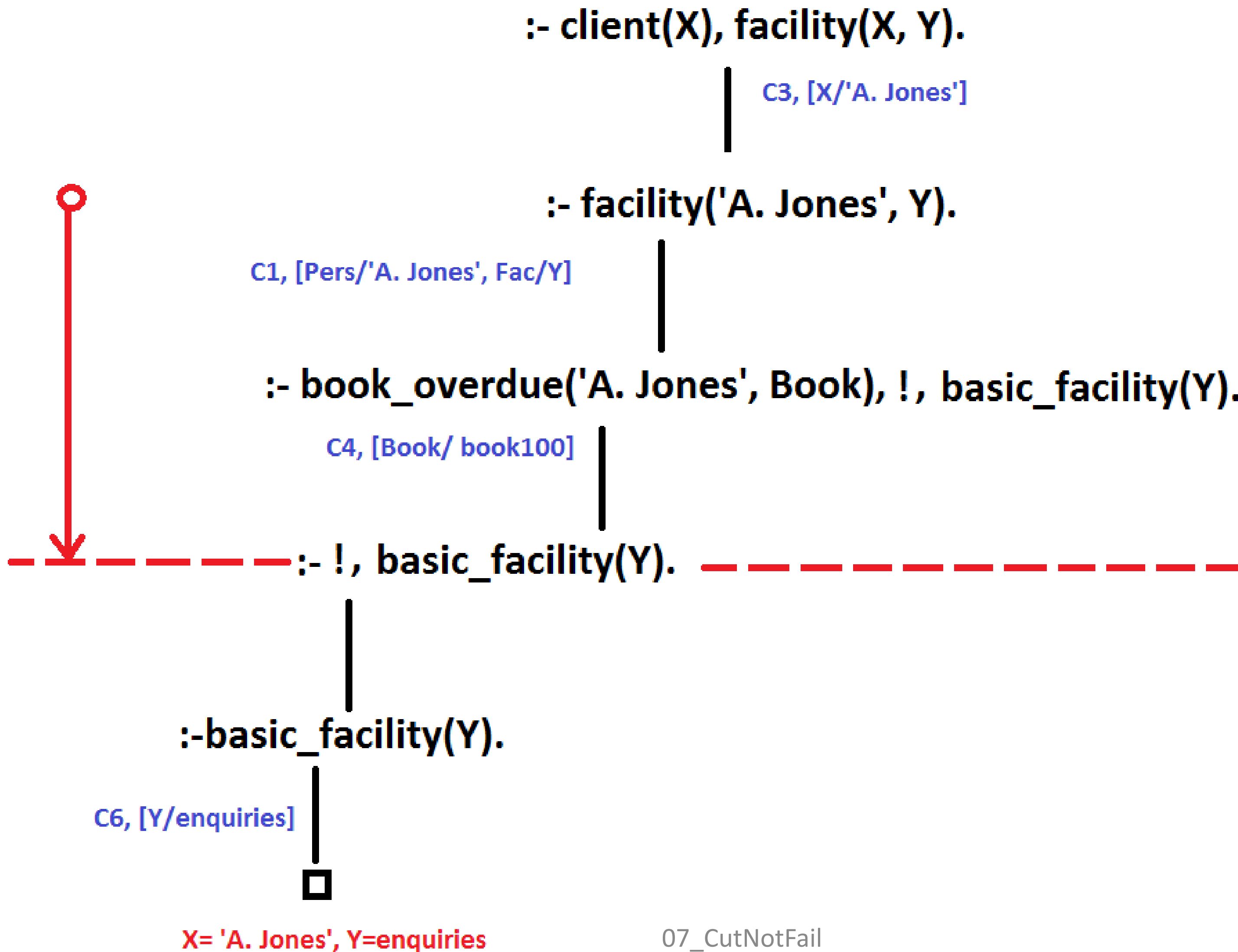
- Effect of cut in this example:

  "If a client is found to have an overdue book, then only allow the client the basic facilities of the library. Don't bother going through all the client's overdue books, and don't consider any other rule about facilities." [Clocksin]

- Formally:

  "When a cut is encountered as a goal, the system thereupon becomes committed to all choices made since the parent goal was invoked. All other alternatives are discarded. Hence an attempt to re-satisfy any goal between the parent goal and the cut goal will fail." [Clocksin]

New Search tree of Library Example



3 reasons to use cut !

1. Confirming the choice of a rule
   “if you get to here, you have picked the correct rule for this goal.” Don't try any other!

2. The cut-fail combination
   “if you get to here, you should stop trying to satisfy this goal.” Return false.

3. Terminating generation of multiple solutions
   “If you get to here, you have found the only solution to this problem.” Don’t try to find alternatives.

Notes

- The cut always has only one meaning to Prolog
  - Instruction about where to and not to backtrack
  - The 3 mentioned, are just to make it easy to apply cut.

- foo:- a, b, c, !, d, e, f.
  - Can backtrack among goals a,b, c
  - Success of c causes the "fence" to be crossed
  - Can backtrack among d, e, f
  - If d fails, no attempt to re-satisfy c, foo will also fail.

Confirming the choice of a rule

- Example:

```
sum_to(1,1):-      ! .
sum_to(N, R):-     N1 is N – 1,
                   sum_to(N1, R1),
                   R is R1 + N.
```

- What will happen if we don’t have ! and enter ; for this query:

```
:- sum_to(1, R).
```

- With !, the second rule will only be used for numbers not equal to 1.
- An alternative form:

```
sum_to(N, 1):-     N =< 1 , ! .
sum_to(N, R):-     N1 is N – 1,
                   sum_to(N1, R1),
                   R is R1 + N.
```

NOT

- An alternative way:

```
sum_to(N, 1):-  N =< 1.
sum_to(N, R):-  \+(N =< 1),
                N1 is N – 1,
                sum_to(N1, R1),
                R is R1 + N.
```

- Not (\+):
  - Good programming style to use \+ instead of !, since easier to read
  - Not always feasible:

a :- b, c. a :- \+b, d. (satisfying b twice)	vs.	a :- b, !, c. a :- d. (satisfying b once→ faster)

Confirming the choice of a rule

- Example: Intersection of A and B

```
C0:  intersection ( [] , B , [] ).
C1:  intersection ( [Ah|At], B, [Ah|Ct]) :-
                                    member ( Ah , B ) , ! ,
                                    intersection ( At , B , Ct ).
C2:  intersection ( [Ah|At], B, C) :-     intersection ( At , B , C ).
```

- Why cut?
  - C1 will apply if head of A is a member of B.
  - C2 will apply if head of A is **not** a member of B, i.e. if the “fence” of ! has not been crossed.
  - Why not a ! for C0? The [Ah|At] will not be unifiable with a [] anyways.

The cut-fail combination

- **fail** is a built-in predicate, w/o any arguments, which always fails.

```
a:- b,c, !, fail.
a:- d.
```

If b and c can be satisfied, a will fail and no more attempts will be made to re-satisfy a.

!, fail. Example

- The average tax payer

```
average_taxpayer(X) :-    foreigner(X), !, fail.
average_taxpayer(X) :-    spouse(X, Y),
                          gross_income(Y, Inc),
                          Inc > 3000,
                          !, fail.
average_taxpayer(X) :-    gross_income(X, Inc),
                          2000 < Inc, 20000> Inc.
```

- Hard to write with not:

```
average_taxpayer(X) :-    \+foreigner(X),
                          \+((spouse(X,Y), gross_income(Y, Inc), Inc>3000)),
                          gross_income(X, Inc),
                          2000 < Inc, 20000> Inc.
```

!, fail. Example

- Actually **not** is defined using fail:

```
not(P):- call(P), !, fail.
not(_).
```

  - ‘call’ queries the database with the predicate P.
  - If P succeeds, not(P) will fail.
  - Otherwise not(P) will succeed.

  - Note that due to Prolog’s method of resolving with the leftmost subgoal, ! will not be reached unless call(P) succeeds.
  - head :- a, b, c.
    b and c won’t be satisfied unless a is satisfied.
    c will not be satisfied unless a and b are satisfied.

Warning!

- not(not(P) is not the same as P!

- Example(1):

```
try1(X) :- member( X, [a, b, c]) .
:- try1(X).
      X = a ;
      X = b ;
      X = c ;
      false.
```

- Example (2):

```
try2(X) :- not( not (member( X, [a, b, c]) ) ).
:- try2(X).
      true.
```

X not instantiated! True for any X!

Terminate a “generate & test” sequence

- Example:

```
divide(N1, N2, Result) :-    is_integer(Result),
                             P1 is Result * N2,
                             P2 is (Result +1) * N2,
                             P1 =< N1, P2 > N1, !.
```

  For example divide(10, 4, 2).

- Generate: Yield all alternatives
  is_integer is the generator of ‘Result’, it generates 0, 1, 2, ...
- Test: Check whether a solution is acceptable
  All other subgoals test if ‘Result’ is acceptable
- CUT: Terminates the generate & test sequence once one solution is found.

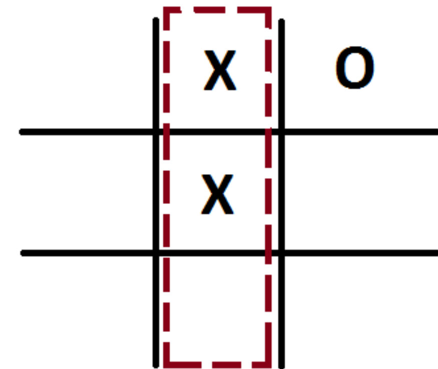
Tic-tac-toe

- Is the o-player forced to put o in a particular position (is x-player threatening to win on its next move?)

b(B1, B2, B3, B4, B5, B6, B7, B8, B9) represents the board
For example: b(e,x,o, e,x,e, e,e,e)

We can define line(B, X,Y,Z) to instantiate X,Y, Z to positions that make up a line:

```
line( b(X,Y,Z,  _,_,_,  _,_,_),  X,Y,Z).
line( b(_,_,_,  X,Y,Z,  _,_,_),  X,Y,Z).
line( b(_,_,_,  _,_,_,  X,Y,Z),  X,Y,Z).
line( b(X,_,_, Y,_,_,  Z,_,_),  X,Y,Z).
...
line( b(X,_,_,  _,Y,_,  _,_,Z),  X,Y,Z).
...
```



Tic-tac-toe (cont.)

```
threatening(e, x, x).
threatening(x, e, x).
threatening(x, x, e).

forced_move(Board) :-   line(Board, X, Y, Z),
                        threatening (X, Y, Z),
                        !.
```

- Generator: line(Board, X, Y, Z) generates possible lines
- Tester: threatening(X, Y, Z) checks if the line has a threatening pattern
- If the generated line is not threatening, *backtracking occurs* and another line is generated
- If threatening does not succeed on any line, "forced_move" will *fail*.
- If threatening succeeds, the "fence" will be crossed, therefore *no backtracking* (no more solutions, one is enough), and it is committed to the instantiated Board

Problems with CUT

- Example (1):

```
append([],  X, X) :- !.
append([H|T], L2, [H|L]) :- append(T, L2, L).

:- append( X, Y, [a, b, c]).
X=[], Y= [a,b,c] ;
```

No more solutions?!

We used ! To indicate correct choice of rule if first list is []
Side effect: no multiple solutions

Problems with CUT (cont.)

- Example (2):

```
no_parents(adam, 0) :- !.
no_parents(eve, 0) :- !.
no_parents(_, 2).
```

:- no_parents (eve, X).
X= 0
Our intention for using cut: No more solutions- works!

:- no_parents(george, X).
X=2

:- no_parents(adam, 2).
true
Side effect: Matches with the last fact!

Problems with CUT (cont.)

```
no_parents(adam, 0) :- !.
no_parents(eve, 0) :- !.
no_parents(_, 2).
```

- Possible modification:

```
no_parents(adam, 0) .
no_parents(eve, 0) .
no_parents(X, 2) :- not(X=adam) , not(X=eve).
```

Will this work with these queries?

```
:- no_parents(X,0).
:- no_parents(X, 2).
:- no_parents(X, Y).
```

Problems with CUT (cont.)

- Example (3):

```
max( X, Y, X) :- X >= Y, !.
max( X, Y, Y).

:- max(1, 10, 10).
true

:- max(1, 10, Z).
Z= 10

:- max(10, 1, Z).
Z= 10

:- max(10, 1, 1).
true          ???
```

Warning!

"If you introduce cuts to obtain correct behaviour when the goals are of one form, there is no guarantee that anything sensible will happen if goals of another form start appearing. " [Clocksin]

- Use cut only if you have a clear policy about the queries and how the rules are going to be used.