EECS 490 – Lecture 20

Logic Programming II

11/16/17

Announcements

- Project 4 due Tue 11/21 at 8pm
- HW5 due 12/5 at 8pm

Review: Horn Clauses

- A logic program is expressed as a set of *axioms* that are assumed to be true
- An axiom takes the form of a *Horn clause*, which specifies a reverse implication:

```
H :- B1, B2, ..., BN
```

Individual *term*

***Head* of clause**

***Body* of clause**

- This is equivalent to

$$(B1 \land B2 \land \ldots \land BN) \Rightarrow H$$

with implicit quantifiers.

Review: Queries

- A *goal* is a query that the system attempts to prove from the axioms

```
parent(P, C) :- mother(P, C).                     % rule 1
parent(P, C) :- father(P, C).                     % rule 2
sibling(A, B) :- parent(P, A), parent(P, B).      % rule 3

mother(molly, bill).                              % fact 1
mother(molly, charlie).                           % fact 2
```

```
   sibling(bill, S)
-> parent(P, bill), parent(P, S)                  (rule 3)
-> mother(P, bill), parent(P, S)                  (rule 1)
-> mother(molly, bill), parent(molly, S)          (fact 1)
-> mother(molly, bill), mother(molly, S)          (rule 1)
-> mother(molly, bill), mother(molly, charlie)    (fact 2)
```

`S = bill` is also a valid solution given the axioms.

Implementing Lists

- Compound terms can represent data structures
- Example: use `pair(A, B)` to represent a pair
  - This won't be a head or body term, so it will be treated as data
- Relations on pairs:

```
cons(A, B, pair(A, B)).
cdr(pair(_, B), B).
car(pair(A, _), A).
is_null(nil).
```

Relates a first and second item to a pair

Anonymous variable

```
?- cons(1, nil, X).
X = pair(1, nil).

?- car(pair(1, pair(2, nil)), X).
X = 1.

?- cdr(pair(1, pair(2, nil)), X).
X = pair(2, nil).

?- cdr(pair(1, pair(2, nil)), X),
   car(X, Y), cdr(X, Z).
X = pair(2, nil), Y = 2, Z = nil.

?- is_null(nil).
true.

?- is_null(pair(1, nil)).
false.
```

Prolog Lists

- Prolog also provides built-in linked lists, specified as elements between square brackets

```
[]    [1, a]    [b, 3, foo(bar)]
```

- The pipe symbol acts like a dot in Scheme, separating some elements from the rest of the list

```
?- writeln([1, 2 | [3, 4]]).
[1,2,3,4]
true.
```

- This allows us to write predicates like the following:

```
contains([X|_], X).
contains([_|Ys], X) :- contains(Ys, X).
```

Numbers and Comparisons

- Prolog includes integer and floating-point numbers
- Comparison predicates can be written in infix order

```
?- 3 =< 4.     % less than or equal
true.

?- 4 =< 3.
false.

?- 3 =:= 3.    % arithmetic equal
true.

?- 3 =\= 3.    % arithmetic not equal
false.
```

- The = operator specifies explicit unification, not equality

Arithmetic

- Arithmetic operators represent compound terms and are not evaluated

```
?- 7 = 3 + 4.
false.
```

7 does not unify with +(3, 4)

- Comparisons perform evaluation on both operands

```
?- 7 =:= 3 + 4.
true.
```

7 is equal to the result of evaluating +(3, 4)

- The `is` operator unifies its first argument with the arithmetic result of its second argument

```
?- 7 is 3 + 4.
true.

?- X is 3 + 4.
X = 7.
```

List Length

- We can now define a predicate for length on our list representation:

```
len(nil, 0).
len(pair(_, B), L) :- len(B, M), L is M + 1.
```

Unify `L` with the result of `+(M, 1)`

Must be second body term so that `M` is sufficiently instantiated for arithmetic

```
?- len(nil, X).
X = 0.

?- len(pair(1, pair(b, nil)), X).
X = 2.
```

- Built-in lists have a built-in `length` predicate

```
?- length([1, a, 3], X).
X = 3.
```

Side Effects

- Prolog provides I/O predicates, including reading from standard input and writing to standard output
- We will only use `write` and `writeln`:

```
?- X = 3, write('The value of X is: '),
   writeln(X).
The value of X is: 3
X = 3.
```

Unification and Search

- A logic solver is built around the processes of *unification* and *search*
- Search in Prolog uses *backward chaining*
  - Start with a set of goal terms
  - Look for a clause whose head can unify with a goal term
  - If unification succeeds, replace the old goal term with the body terms of the clause
  - Search succeeds when no more goal terms remain
- Unification attempts to unify two terms, which may require recursively unifying subterms
  - May require *instantiating* variables to values

Unification

- An atomic term only unifies with itself
- A variable unifies with any term
  - If the other term is not a variable, then the variable is *instantiated* with the value of the other term, i.e. all occurrences of the variable are replaced with the value
  - If the other term is a variable, the two variables are bound together such that later instantiating one with a value also instantiates the other with the same value
- A compound term unifies with another compound term if the functors and number of arguments are the same, and the arguments recursively unify

```
X = 3
Y = foo(1, Z)
foo(1, A) = foo(B, 3)  % unifies B = 1, A = 3
```

Instantiation and Renaming

- Applying a clause involves renaming variables that occur in different contexts to be unique and can result in instantiation of variables
  - Analogous to α- and β-reduction in λ-calculus
- Example:

```
foo(X, Y) :- bar(Y, X).
```

```
?- foo(3, X).
```

1. Rename rule to `foo(X1, Y1) :- bar(Y1, X1).`
2. Unify `foo(3, X)` with `foo(X1, Y1)`, resulting in `X1 = 3` and `X <=> Y1`
3. New goal term `bar(X, 3)`

Search Order

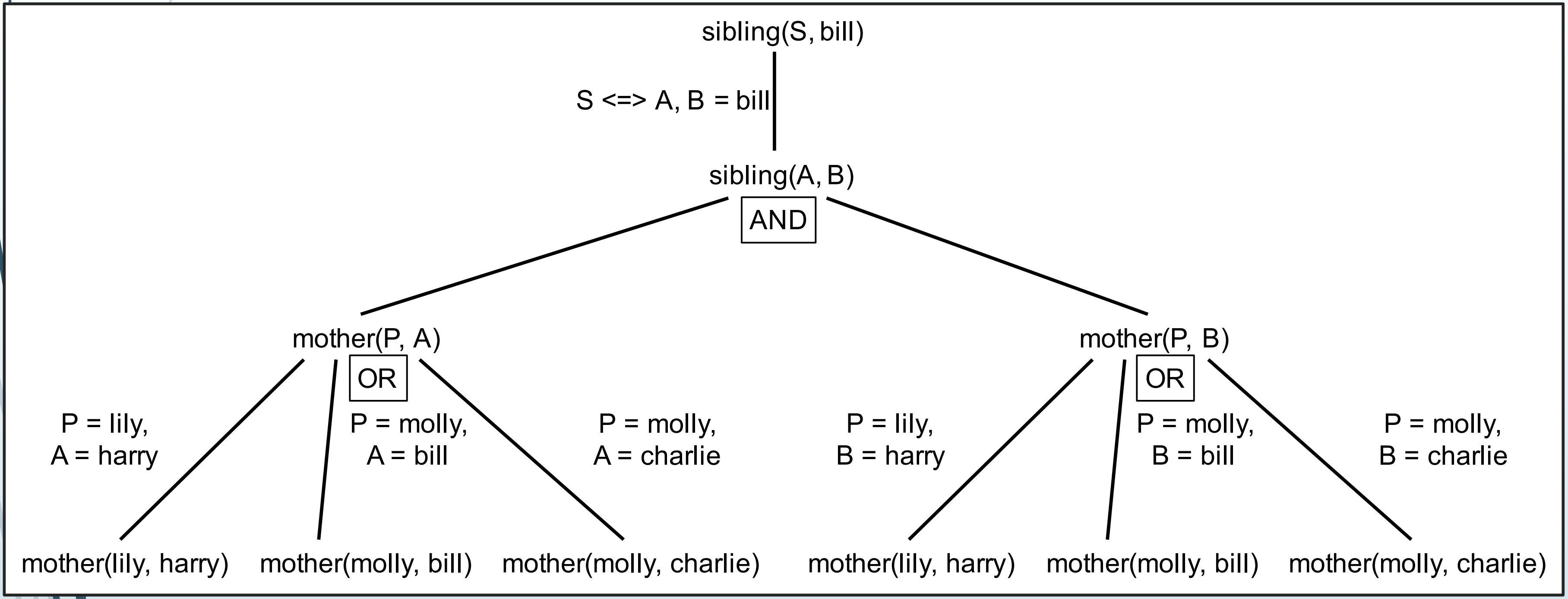
- In pure logic programming, search order is irrelevant as long as the search terminates
- In Prolog, clauses are applied in program order, and terms within a body are resolved in left-to-right order
- Example:

```
sibling(A, B) :- mother(P, A), mother(P, B).

mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

```
?- sibling(S, bill)
S = bill
```

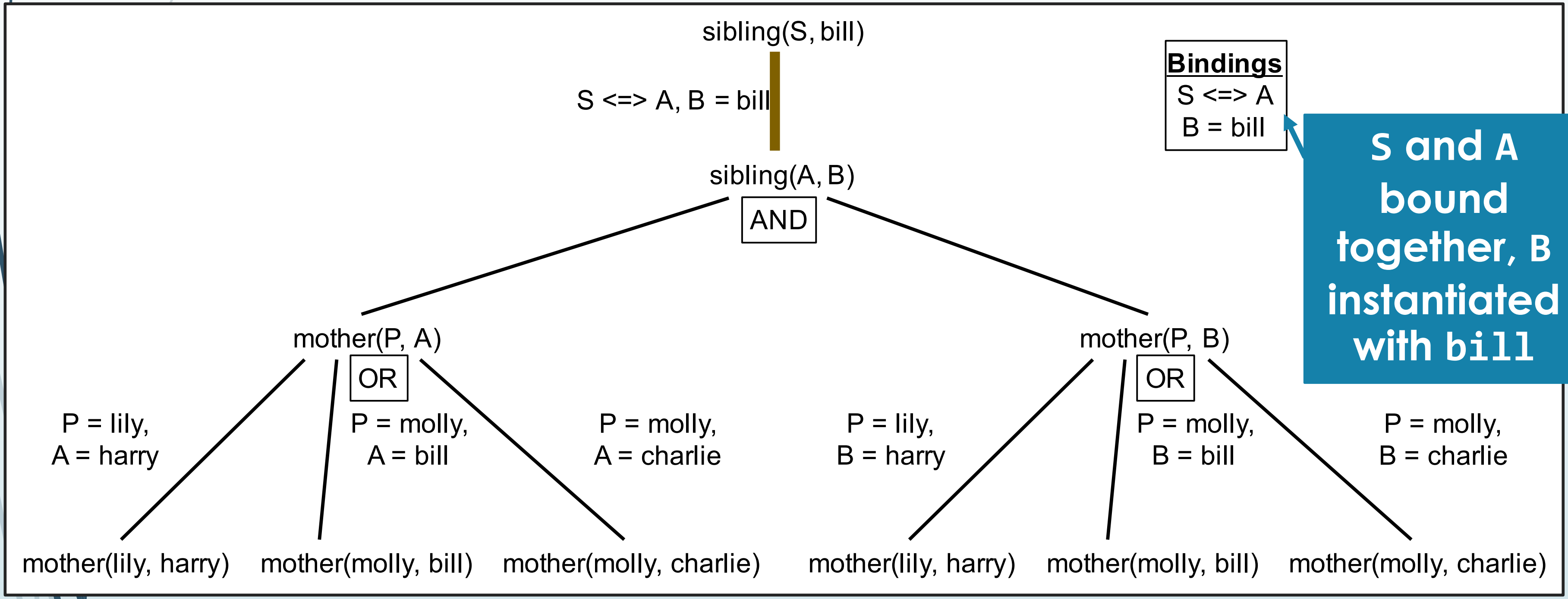
Search Tree

- Search encounters choice points, and backtracking is required on failure or if the user asks for more solutions



Search Tree

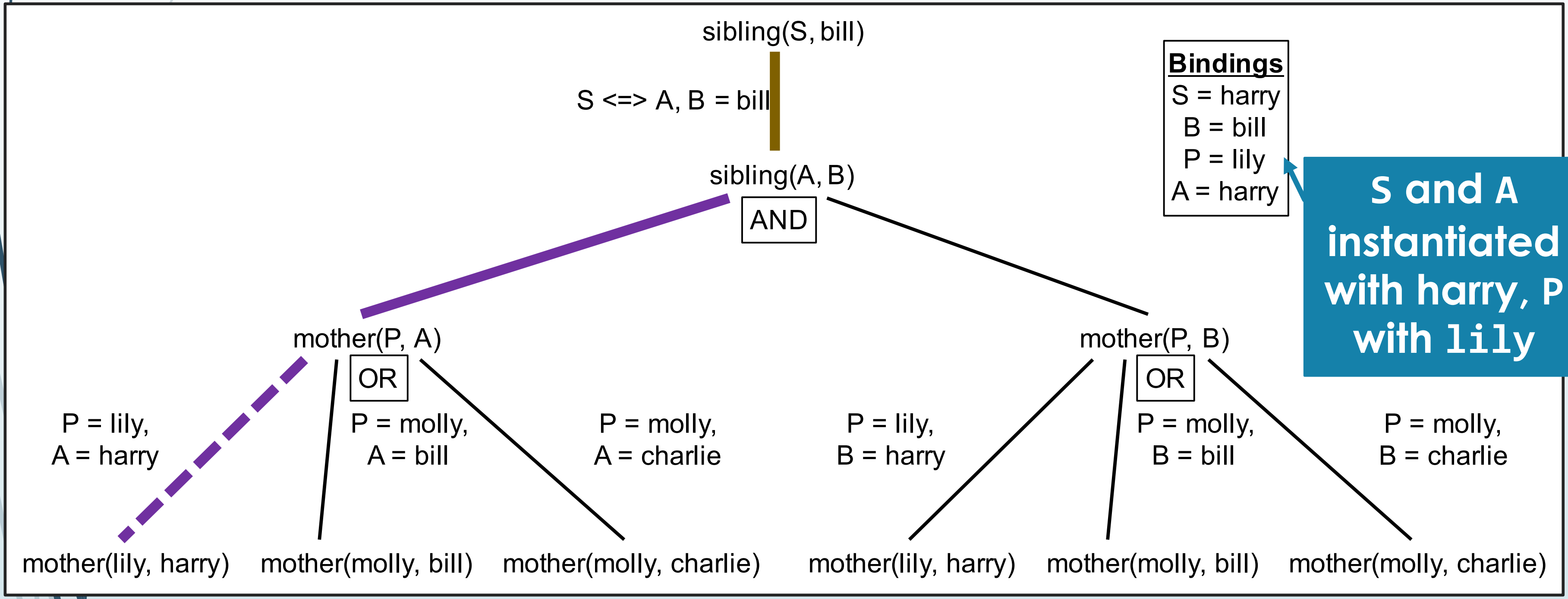
- First, `sibling(S, bill)` is unified with the head term `sibling(A, B)`, and the body terms of the clause are added to the goals



S and A bound together, B instantiated with `bill`

Search Tree

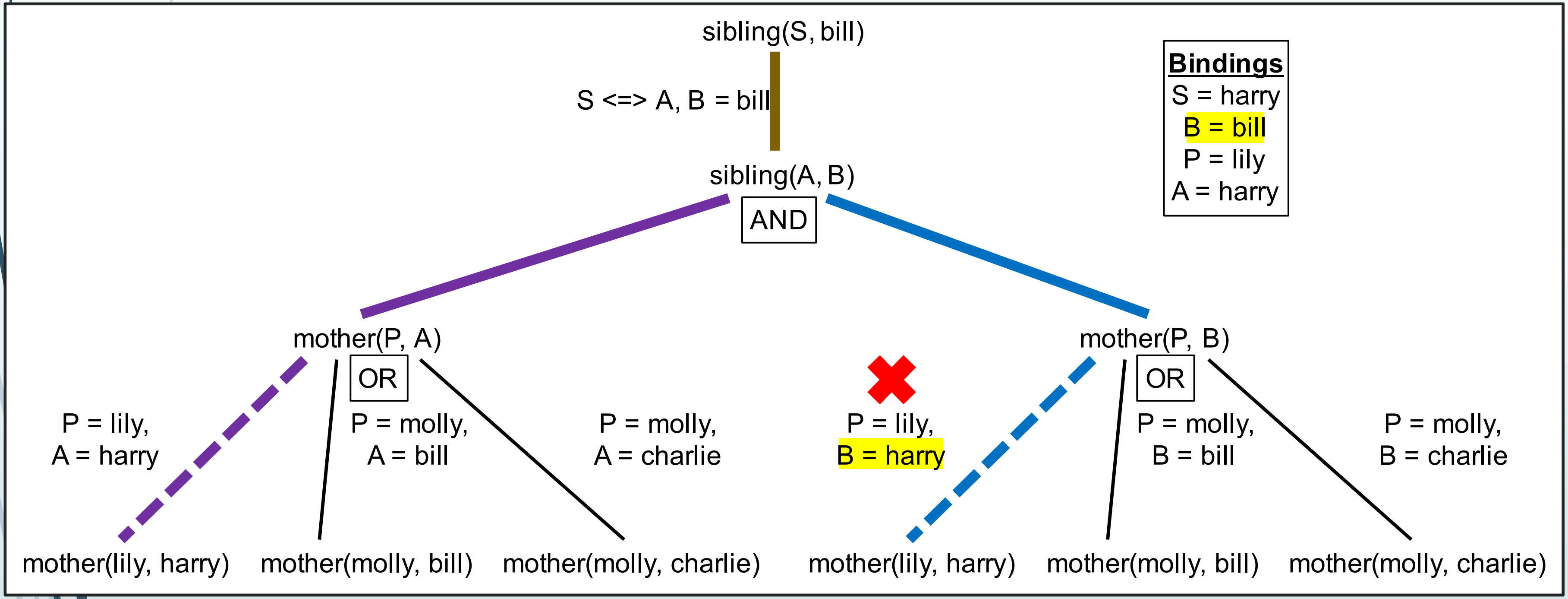
- The goal `mother(P, A)` is solved first, with an initial choice of applying the fact `mother(lily, harry)`



S and A instantiated with harry, P with `lily`

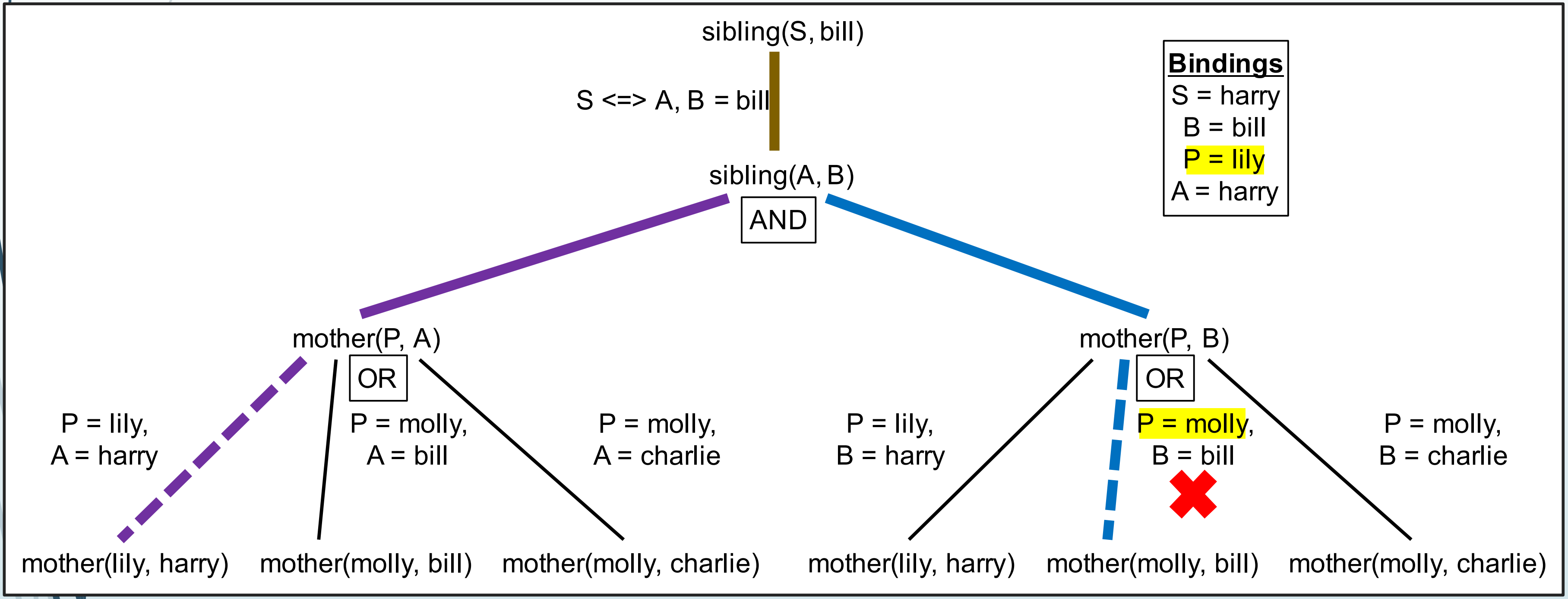
Search Tree

- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



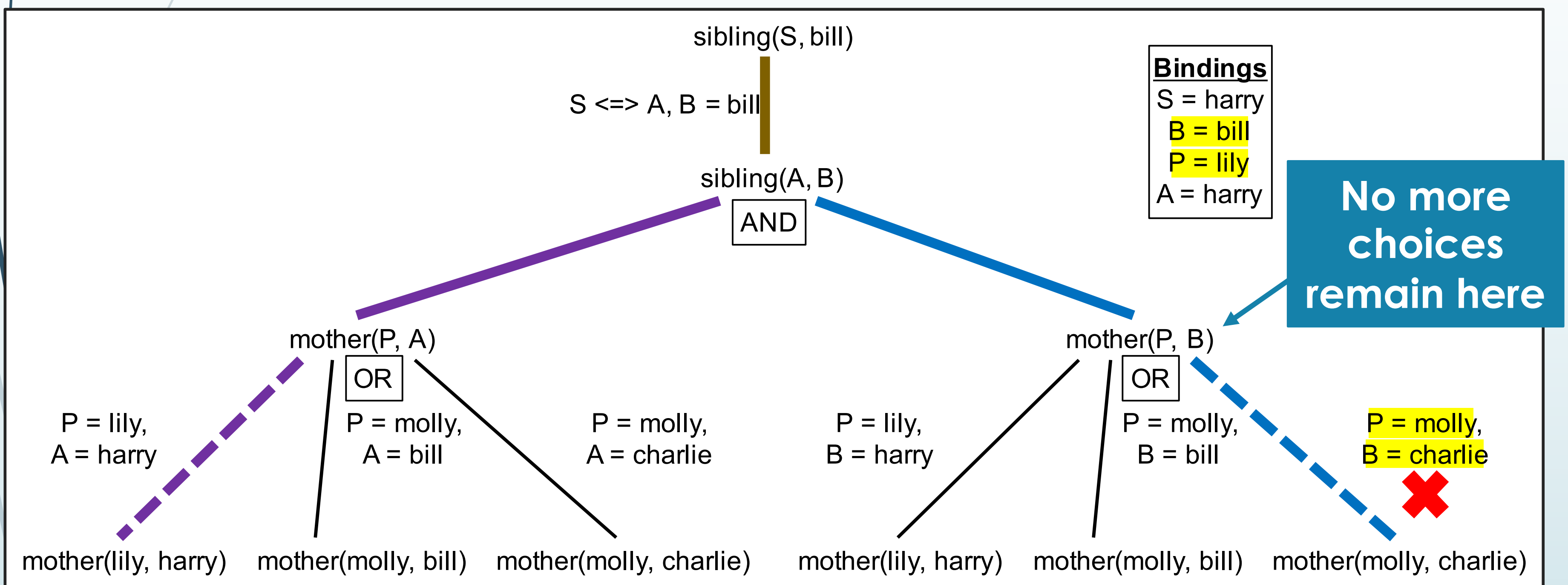
Backtracking

- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- However, unification of `P = lily` with `molly` fails



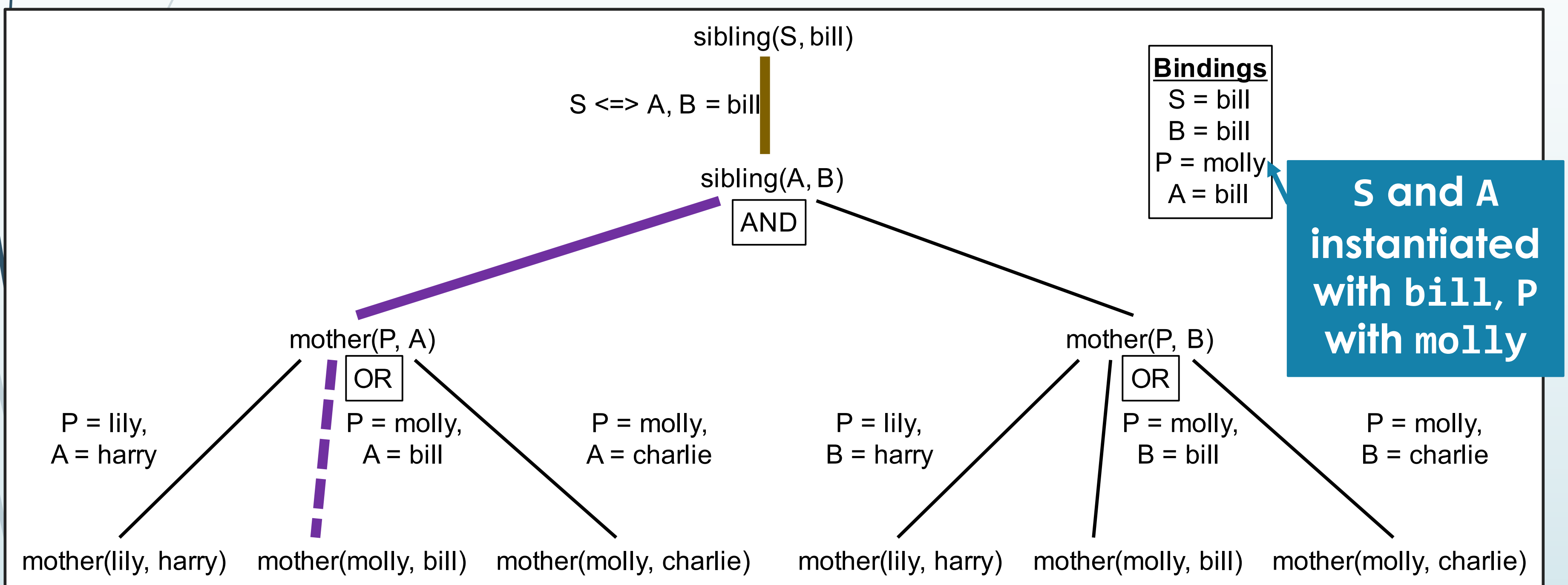
Backtracking

- The search backtracks once again, attempting to apply the fact `mother(molly, charlie)`
- However, unification of `P = lily` with `molly` fails



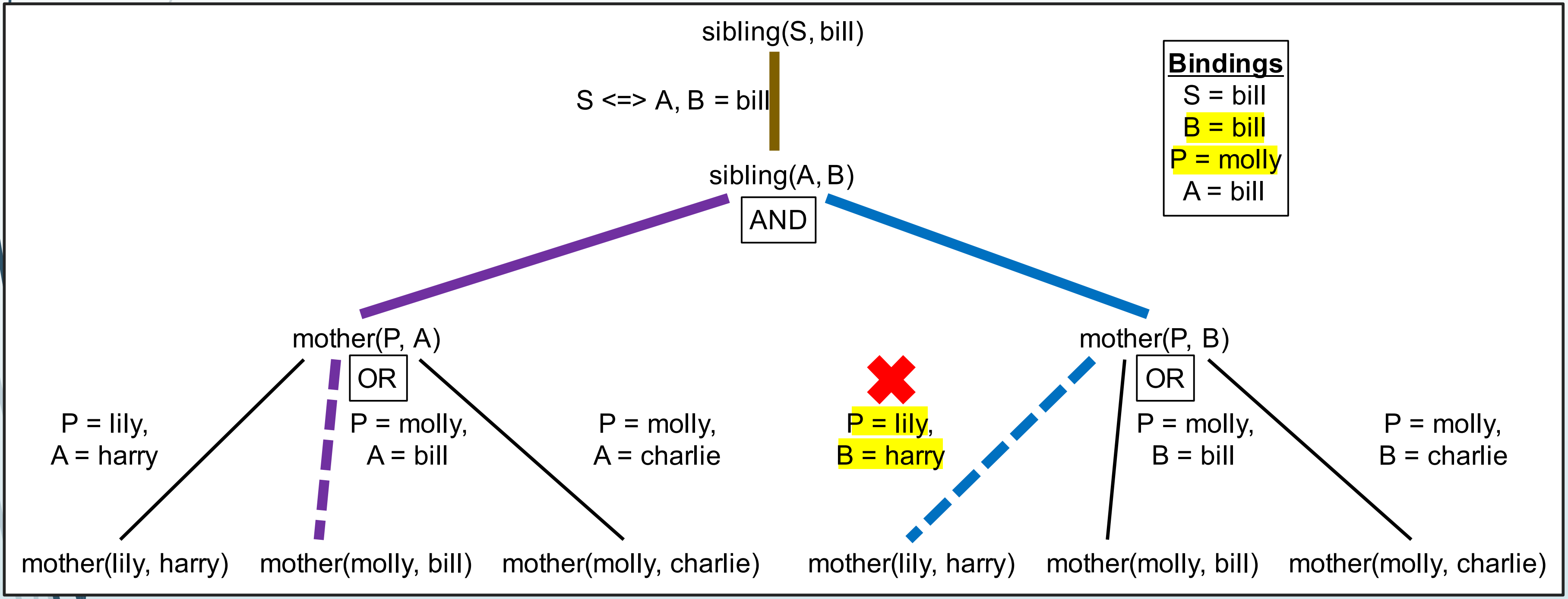
Backtracking

- The search backtracks to the preceding choice point, unifying `mother(P, A)` with `mother(molly, bill)`



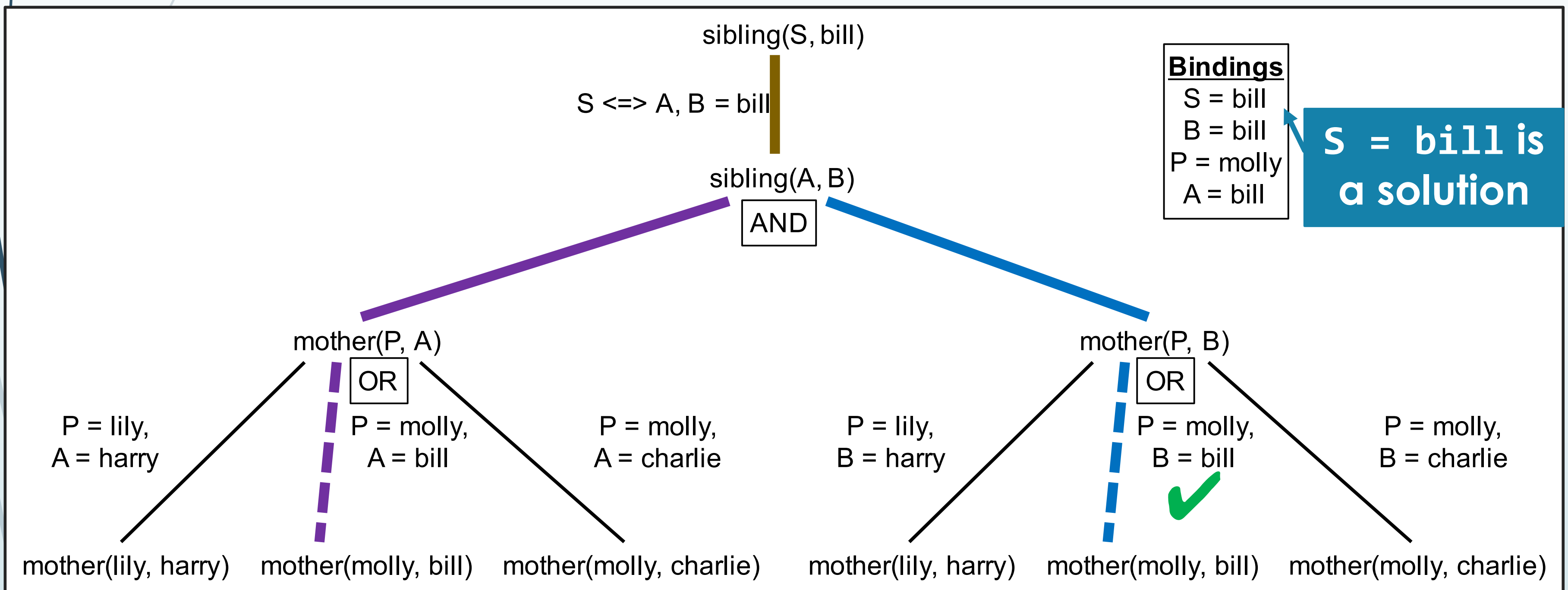
Search Tree

- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



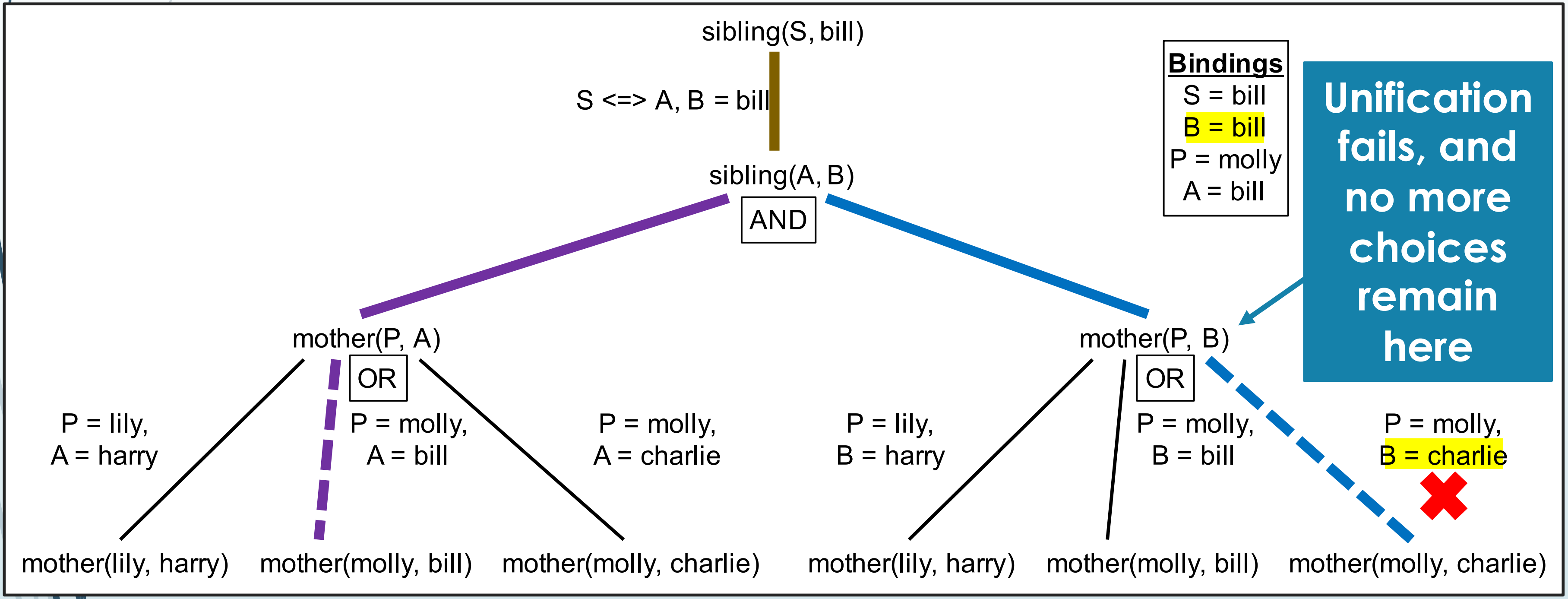
First Solution

- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- Unification succeeds, and no goal terms remain



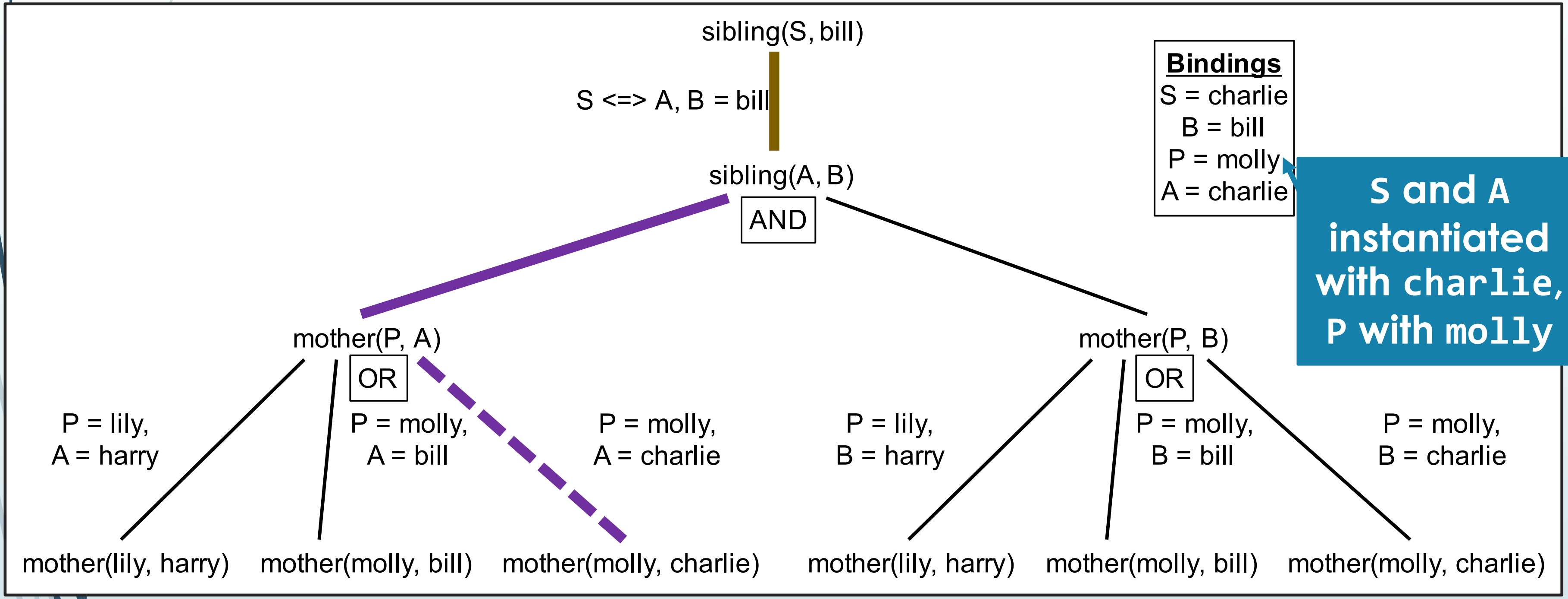
Continuing the Search

- If we ask the interpreter for another solution, it backtracks to the previous choice point, attempting to apply the fact `mother(molly, charlie)`



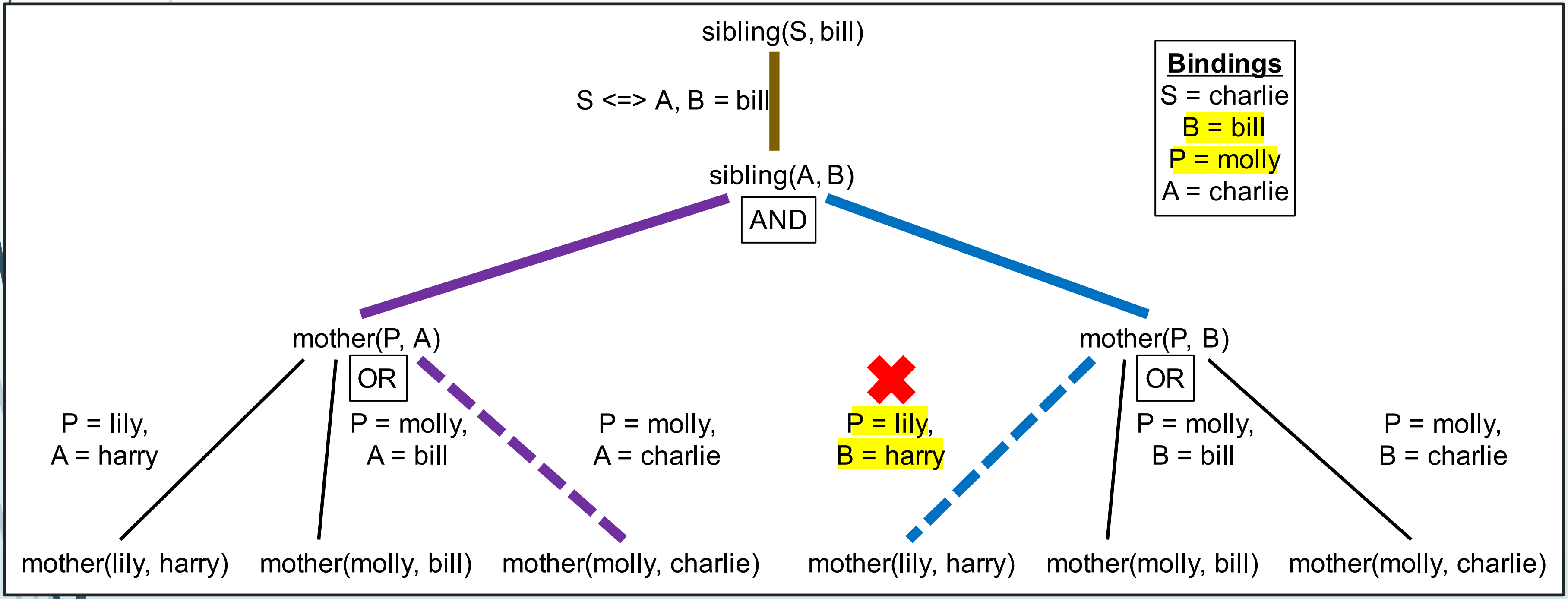
Backtracking

- The search backtracks to the preceding choice point, unifying `mother(P, A)` with `mother(molly, charlie)`



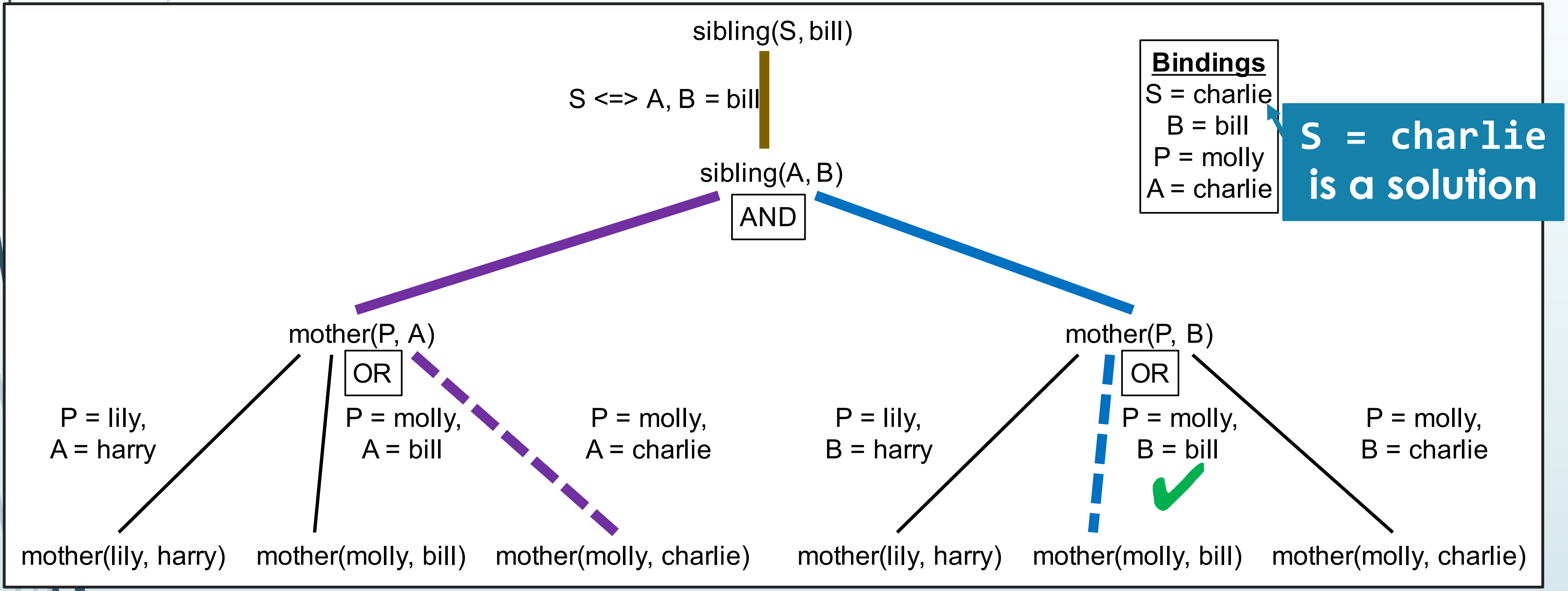
Search Tree

- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



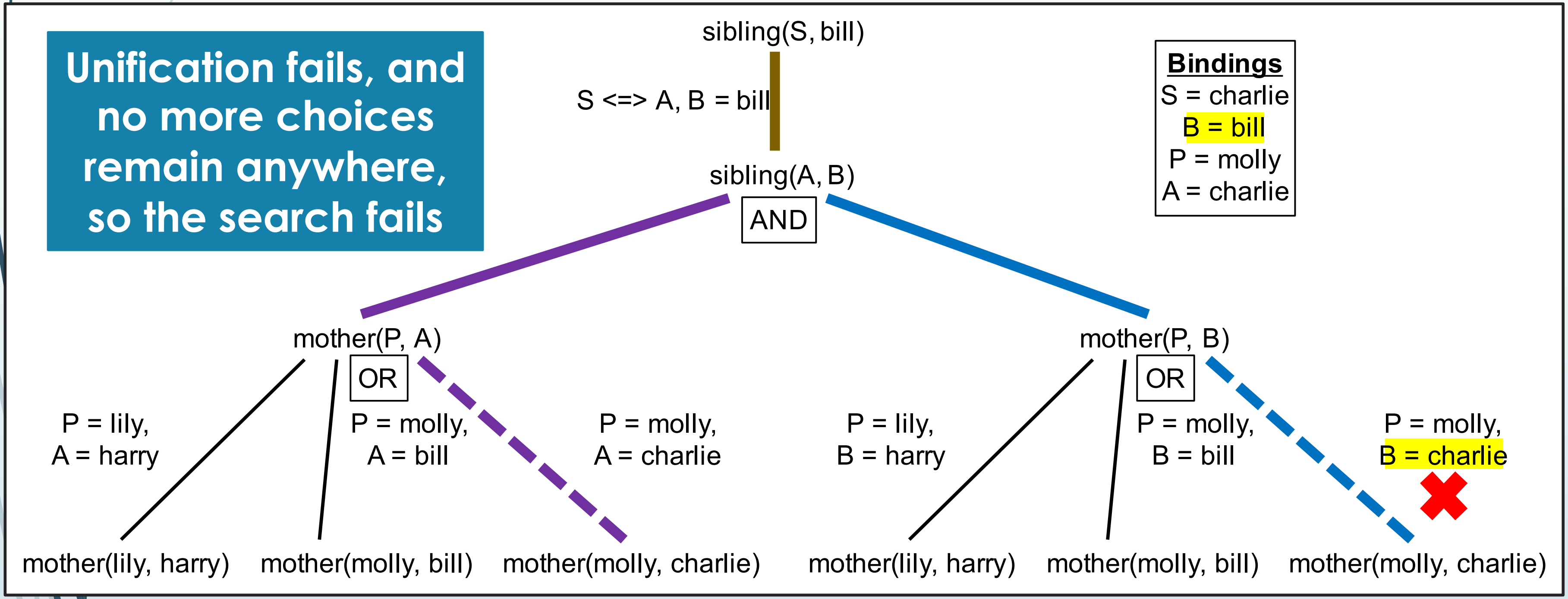
Second Solution

- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- Unification succeeds, and no goal terms remain



No More Solutions

- If we ask the interpreter for another solution, it backtracks to the previous choice point, attempting to apply the fact `mother(molly, charlie)`



Cut Operator

- The cut operator (`!`) tells the search engine to eliminate choice points associated with the current predicate
- However, this can cause some queries to fail, as it prevents backtracking from considering other choices:

```
contains([Item|_], Item) :- !.
contains([_|Rest], Item) :-
  contains(Rest, Item).
```

```
?- contains([1, 2, 3, 4], X), X = 3.
false.
```

- We will only use the cut operator in a query to restrict ourselves to the first solution; we will **not** use it in a rule

Negation

- Prolog provides limited negation operators
  - Explicit negation: \+
  - Negation of unification: \=

- We can try to rewrite the `sibling` rule to avoid the result that `bill` is his own sibling in `sibling(S, bill)`:

```
sibling(A, B) :- A \= B,
  mother(P, A), mother(P, B).
```

Variable `A` `<=>` `S` unifies with anything, so negation always fails

- Instead, write it as:

```
sibling(A, B) :- mother(P, A), mother(P, B),
  A \= B.
```

Variables `A` and `B` now instantiated, so it only fails when `A = bill` and `B = bill`

Limits of Negation

- If we query whether `harry` and `bill` are not siblings, the query succeeds:

```
?- \+(sibling(harry, bill)).
true.
```

- But if we attempt to find someone who is not a sibling of `bill`, the query fails:

```
?- \+(sibling(S, bill)).
false.
```

There is a solution to `sibling(S, bill)`, so the negation fails

- Negation is defined as attempting to prove what is being negated, and if the proof fails, the negation is true

- This limit is fundamental to logic programming, which does not provide the full power of first-order predicate calculus

Break Time!

- We'll start again in five minutes.

Example: Digits

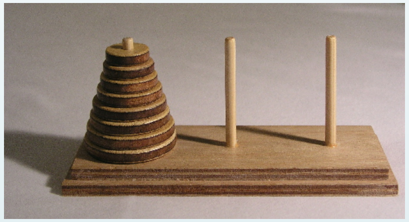
- Find a 5 digit number whose first digit counts the number of 0s, second counts the number of 1s, etc.

```
count(_, [], 0).
count(Item, [Item|Rest], Count) :-
  count(Item, Rest, RestCount),
  Count is RestCount + 1.
count(Item, [Other|Rest], Count) :-
  Item =\= Other,
  count(Item, Rest, Count).

is_digit(0).  is_digit(1).  is_digit(2).
is_digit(3).  is_digit(4).

digits(M) :-
  M = [N0, N1, N2, N3, N4],
  is_digit(N0), is_digit(N1), is_digit(N2),
  is_digit(N3), is_digit(N4),
  count(0, M, N0), count(1, M, N1),
  count(2, M, N2), count(3, M, N3),
  count(4, M, N4).
```

Example: Tower of Hanoi

- Move *N* discs from one rod to another, using a third rod as temporary storage
- The discs have varying size, and cannot place a larger disc on a smaller one
- Print a move:

```
move(Disc, Source, Target) :-
  write('Move disc '), write(Disc),
  write(' from '), write(Source),
  write(' to '), writeln(Target).
```



- Solve the puzzle:

```
hanoi(1, Source, Target, _) :-
  move(1, Source, Target).
hanoi(N, Source, Target, Temp) :-
  M is N - 1,
  hanoi(M, Source, Temp, Target),
  move(N, Source, Target),
  hanoi(M, Temp, Target, Source).
```

Example: Quicksort

- Partition:

```
partition(_, [], [], []).
partition(Pivot, [Item|Rest], [Item|Less], NotLess) :-
  Item < Pivot,
  partition(Pivot, Rest, Less, NotLess).
partition(Pivot, [Item|Rest], Less, [Item|NotLess]) :-
  Item >= Pivot,
  partition(Pivot, Rest, Less, NotLess).
```

- Sort:

```
quicksort([], []).
quicksort([Item|Rest], Sorted) :-
  partition(Item, Rest, Less, NotLess),
  quicksort(Less, SortedLess),
  quicksort(NotLess, SortedNotLess),
  append(SortedLess, [Item|SortedNotLess], Sorted).
```

Example: Primes

- Sieve of Eratosthenes:

```
numbers(2, [2]).
numbers(Limit, Numbers) :-
  M is Limit - 1, numbers(M, NumbersToM),
  append(NumbersToM, [Limit], Numbers).

is_not_multiple(N, D) :- R is mod(N, D), R =\= 0.

filter_not_multiple(_, [], []).
filter_not_multiple(Factor, [First|Rest],
                    [First|FilteredRest]) :-
  is_not_multiple(First, Factor),
  filter_not_multiple(Factor, Rest, FilteredRest).
filter_not_multiple(Factor, [_|Rest], FilteredRest) :-
  filter_not_multiple(Factor, Rest, FilteredRest).

sieve([]).
sieve([First|Rest], [First|SievedRest]) :-
  filter_not_multiple(First, Rest, FilteredRest),
  sieve(FilteredRest, SievedRest).

primes(Limit, Primes) :-
  numbers(Limit, Numbers), sieve(Numbers, Primes).
```