EECS 390 – Lecture 18

Logic Programming II

3/27/24

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:

$$\texttt{H :- } \underbrace{\texttt{B}_1\texttt{, B}_2\texttt{, ..., B}_N}_{\text{Body of clause}}$$

Individual *term*

***Head* of clause**

***Body* of clause**

- This is equivalent to

$$(B_1 \wedge B_2 \wedge \ldots \wedge B_N) \Rightarrow H$$

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
```

- Possible reasoning:

Goal

```
   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.

Prolog

- Prolog is the foundational language of logic programming and is the most widely used
- A Prolog program consists of a set of Horn clauses, using the syntax on the preceding slides
- A Horn clause has a head term and optional body terms
- A term may be atomic, compound, or a variable
  - ***Atomic***: atoms and numbers
    - ***Atom***: Scheme-like symbol or quoted string
    - If an atom starts with a letter, it must be lowercase

```
hello    =<    +    'logic programming'
```

  - ***Variables***: symbols that start with an uppercase letter

```
Hello    X
```

Compound Terms

- A compound term consists of a ***functor***, which is an atom, followed by a list of one or more argument terms

```
pair(1, 2)  wizard(harry)  writeln(hello(world))
```

- A compound term is interpreted as a ***predicate***, with a truth value, if it is a head term, a body term, or the goal

- Otherwise, the compound term is interpreted as data
  - e.g. `hello(world)` in `writeln(hello(world))`

Facts and Rules

- A Horn clause with no body is a ***fact***, since it is always true

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

Period signifies end of clause

- A Horn clause with a body is a ***rule***

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

  - Meaning:
    - If `mother(P, C)` is true, then so is `parent(P, C)`
    - If `parent(P, A)` and `parent(P, B)` are true, then so is `sibling(A, B)`
- A program is a set of Horn clauses

Goals and Queries

- A ***goal*** is a predicate that the interpreter attempts to prove
- Loading the program from the previous slide and entering the goal `sibling(bill, S)` produces:

```
?- sibling(bill, S).
S = bill ;
S = charlie.
```

Ask for more solutions

- A semicolon asks for more solutions
- A period ends a query
  - Can be entered by the user
  - Can be produced by the interpreter, in which case it is certain no more solutions exist

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.
```

Singleton Variables

- A ***singleton variable*** is a variable that only appears once in an axiom
- Singleton variables can occur inadvertently as a result of a typo:

Oops

```
cons(First, Second, pair(Frist, Second)).
```

- To address this, the Prolog interpreter warns about the occurrence of a singleton variable
- We can inform the interpreter about an intentional singleton by using a name that begins with an underscore

Named, intentional singleton variable

```
cdr(pair(_First, Second), Second).
car(pair(First, _), First).
```

Anonymous variable – does not match any other occurrence of _

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

Requires the first argument to be a list of at least one item

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(_, Second), Length) :-
    len(Second, SLength), Length is SLength + 1.
```

Unify `Length` with the result of `+(SLength, 1)`

Must be second body term so that `SLength` 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 (or an uninstantiated variable)
- An uninstantiated 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
```

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:

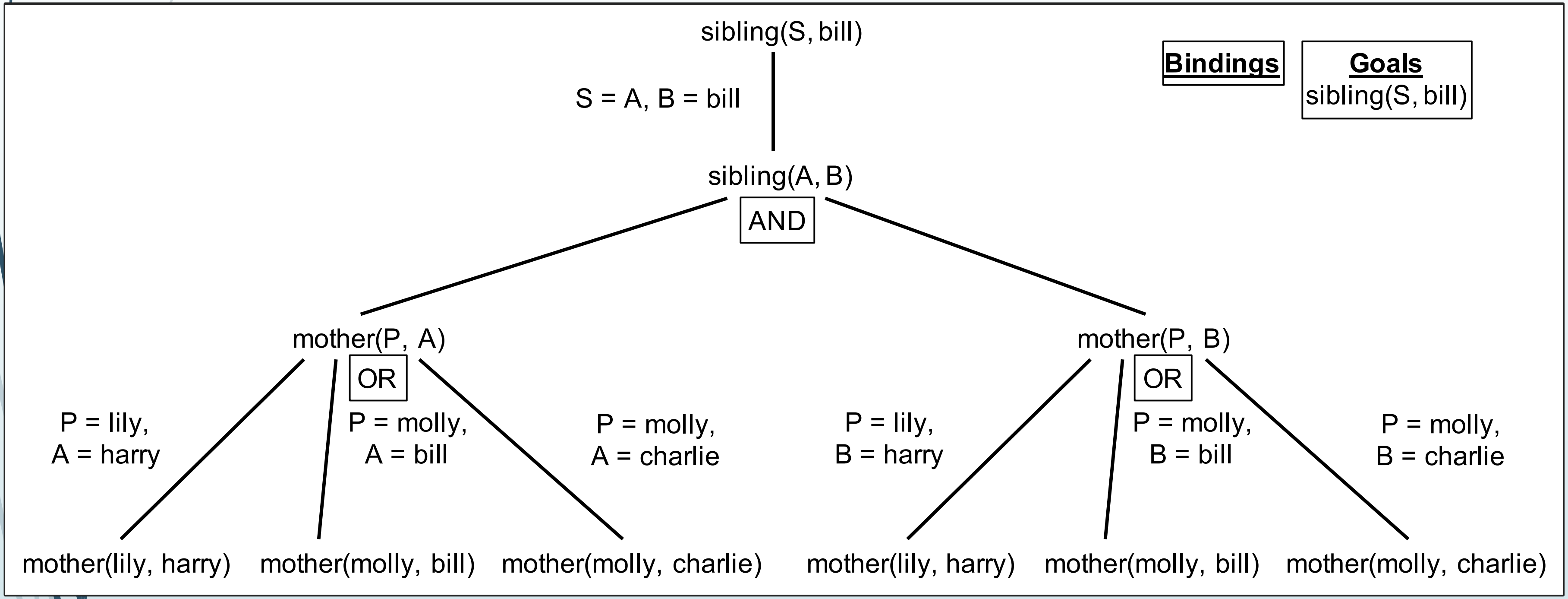
```
sibling(A, B) :-
  mother(P, A), mother(P, B).

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

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

Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

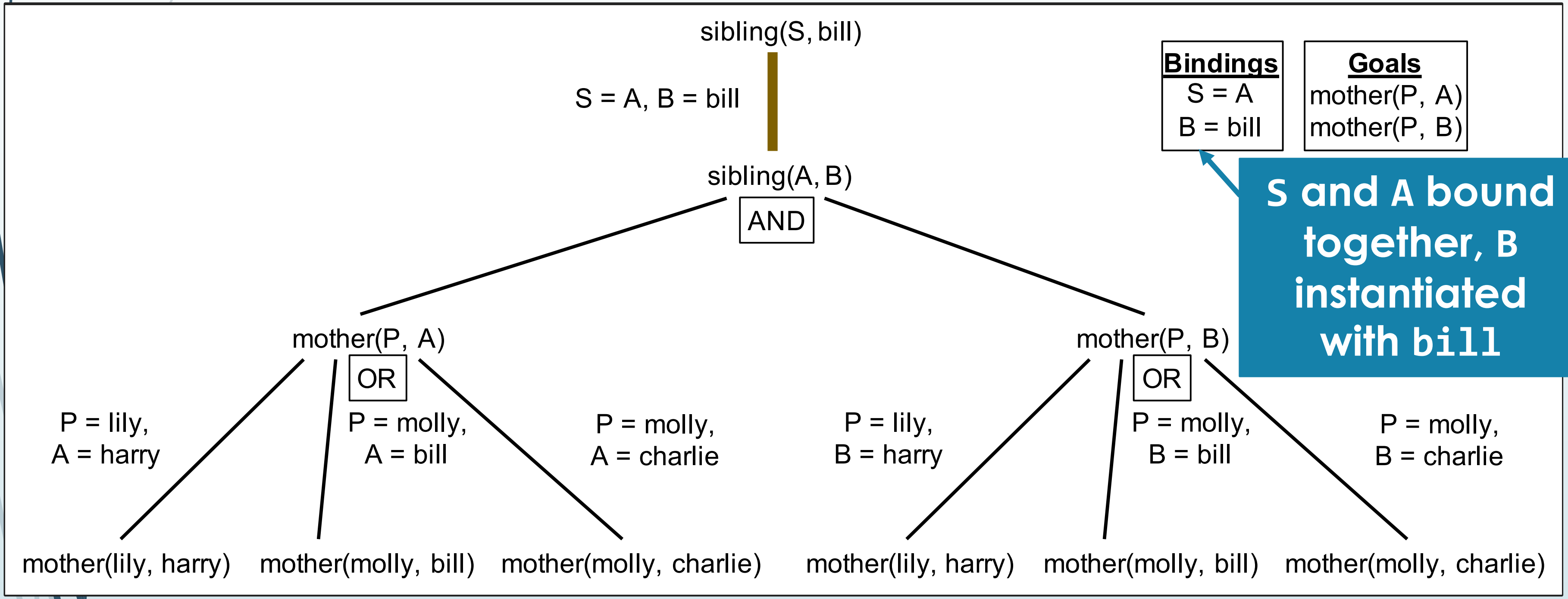
- Search encounters choice points, and backtracking is required on failure or if the user asks for more solutions



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

- 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

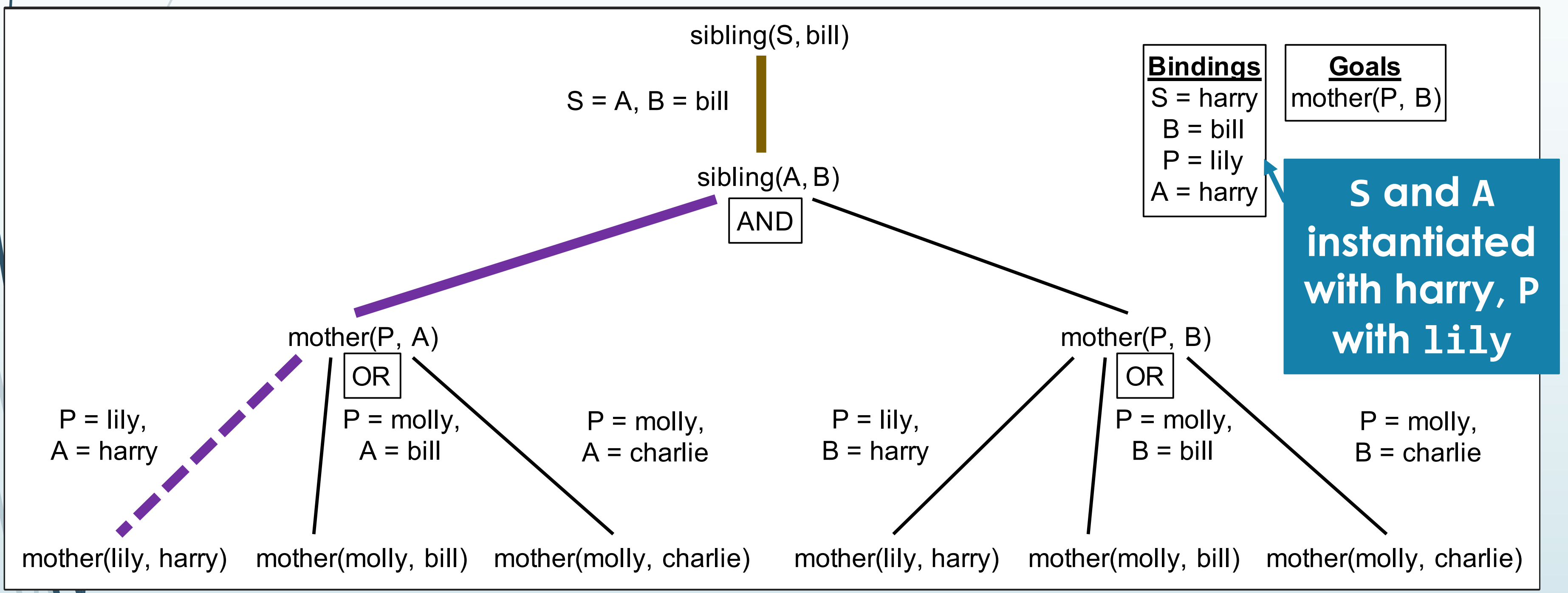


S and A bound together, B instantiated with bill

Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

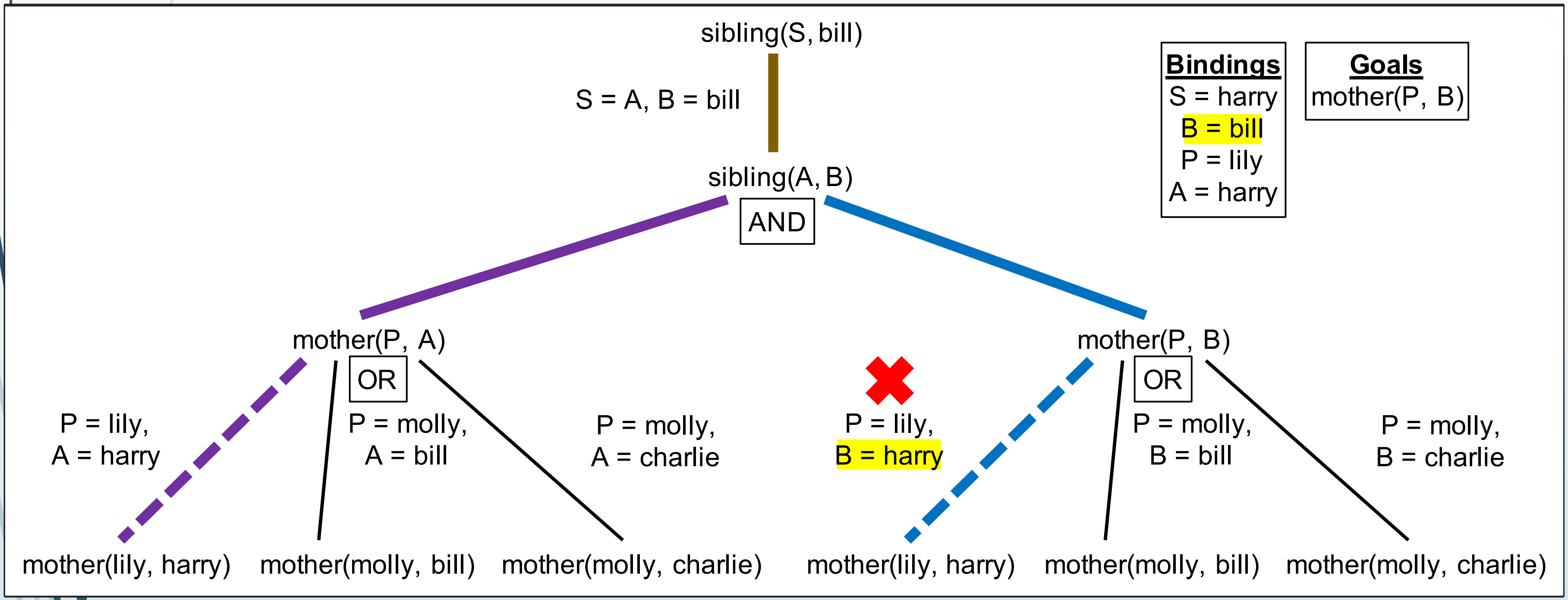
- The goal `mother(P, A)` is solved first, with an initial choice of applying the fact `mother(lily, harry)`



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

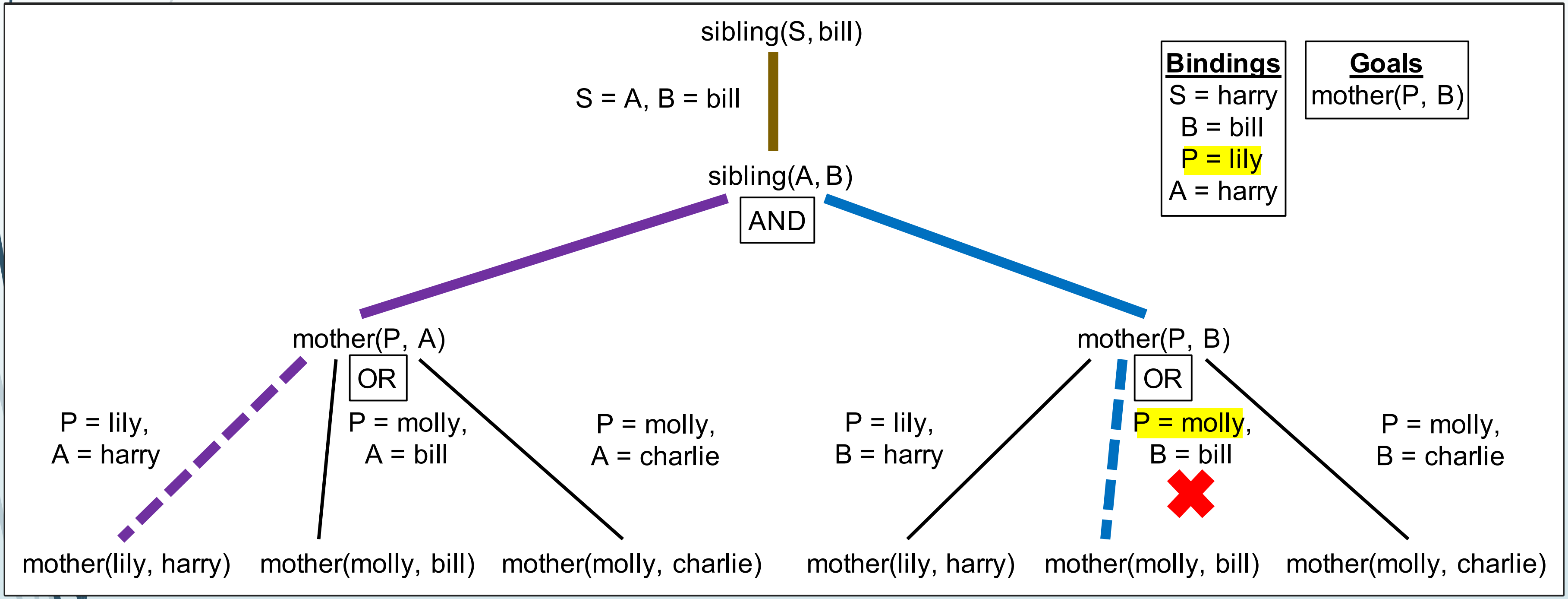
- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



Backtracking

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

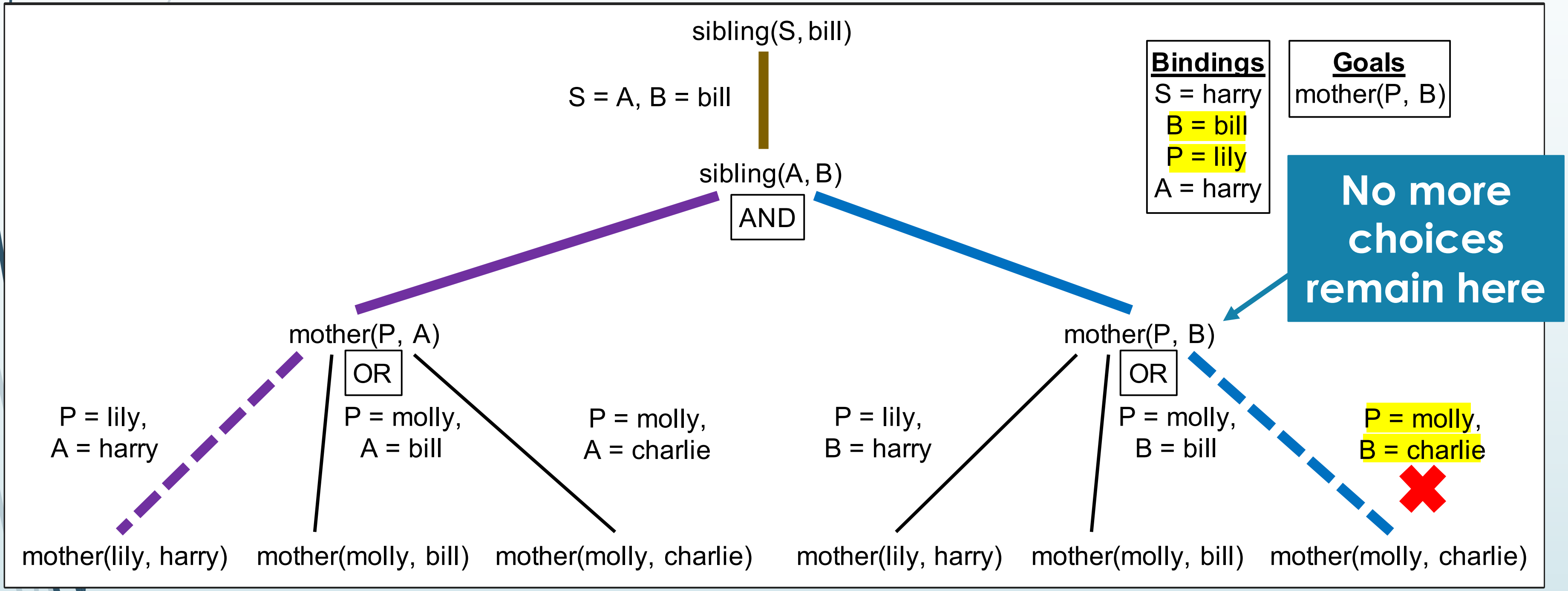
- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- However, unification of `P = lily` with `molly` fails



Backtracking

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

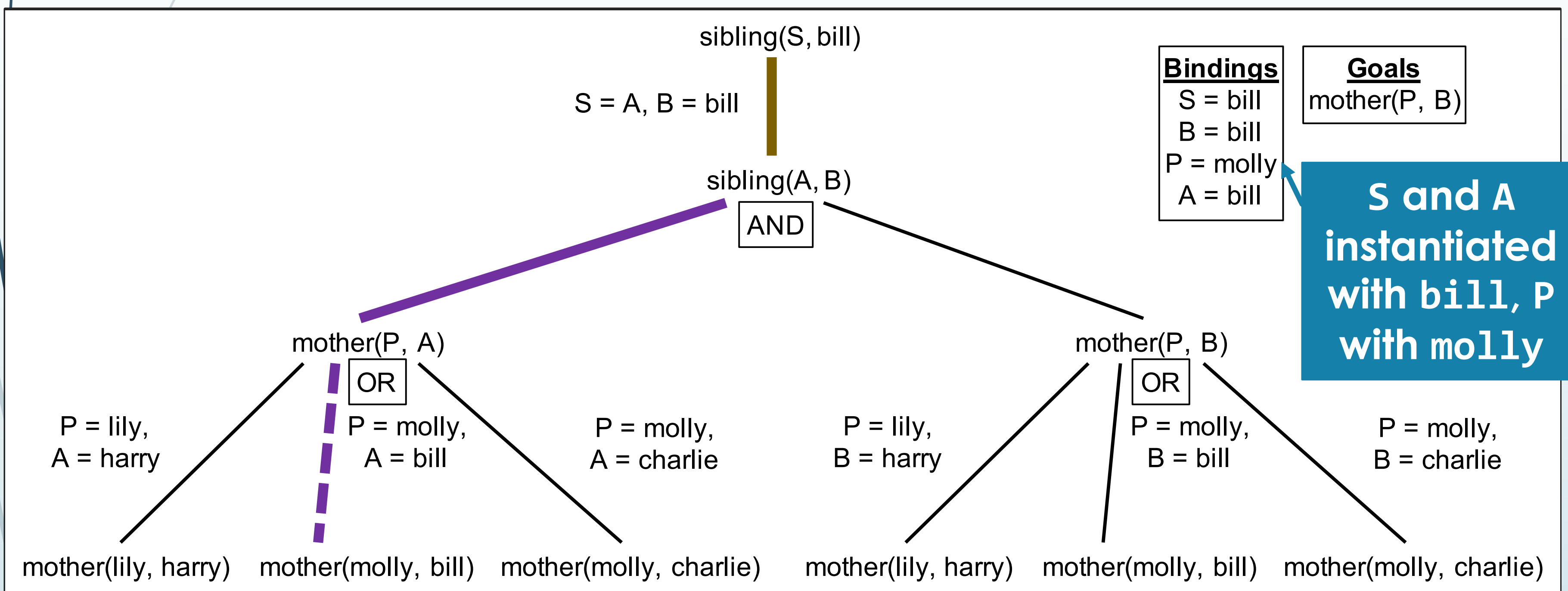
- The search backtracks once again, attempting to apply the fact `mother(molly, charlie)`
- However, unification of `P = lily` with `molly` fails



Backtracking

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

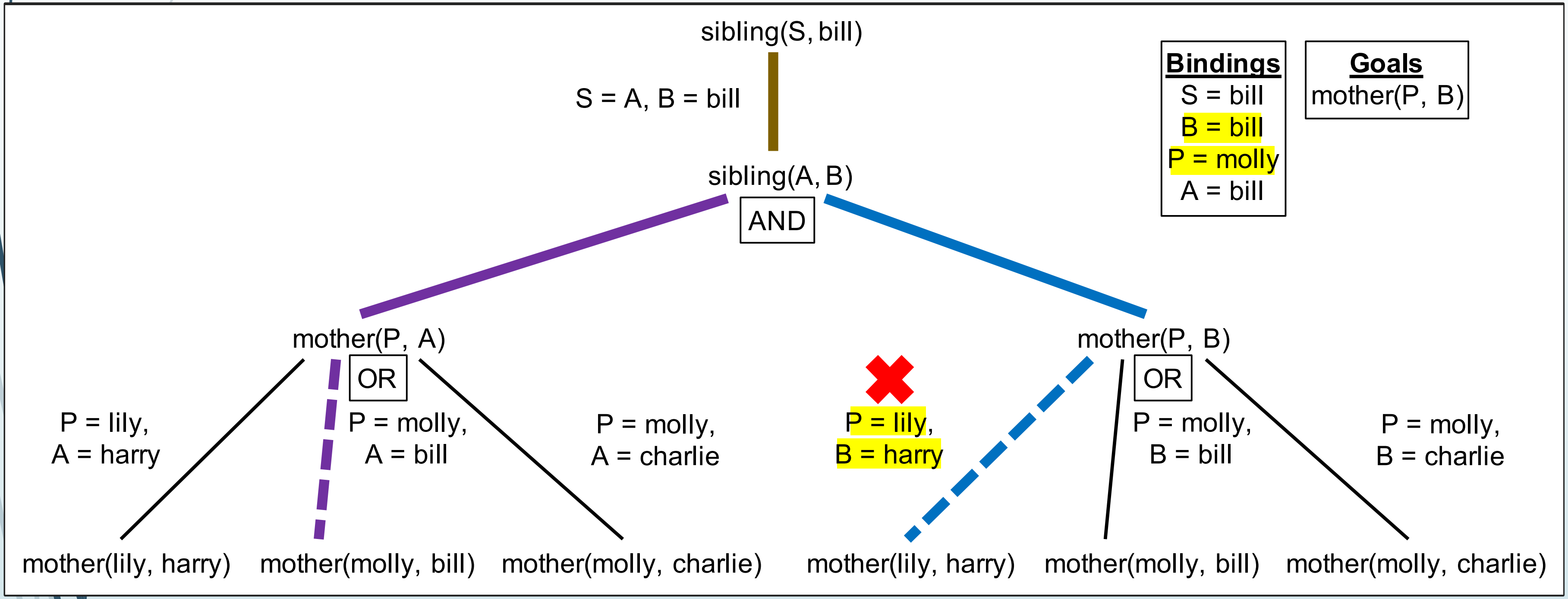
- The search backtracks to the preceding choice point, unifying `mother(P, A)` with `mother(molly, bill)`



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

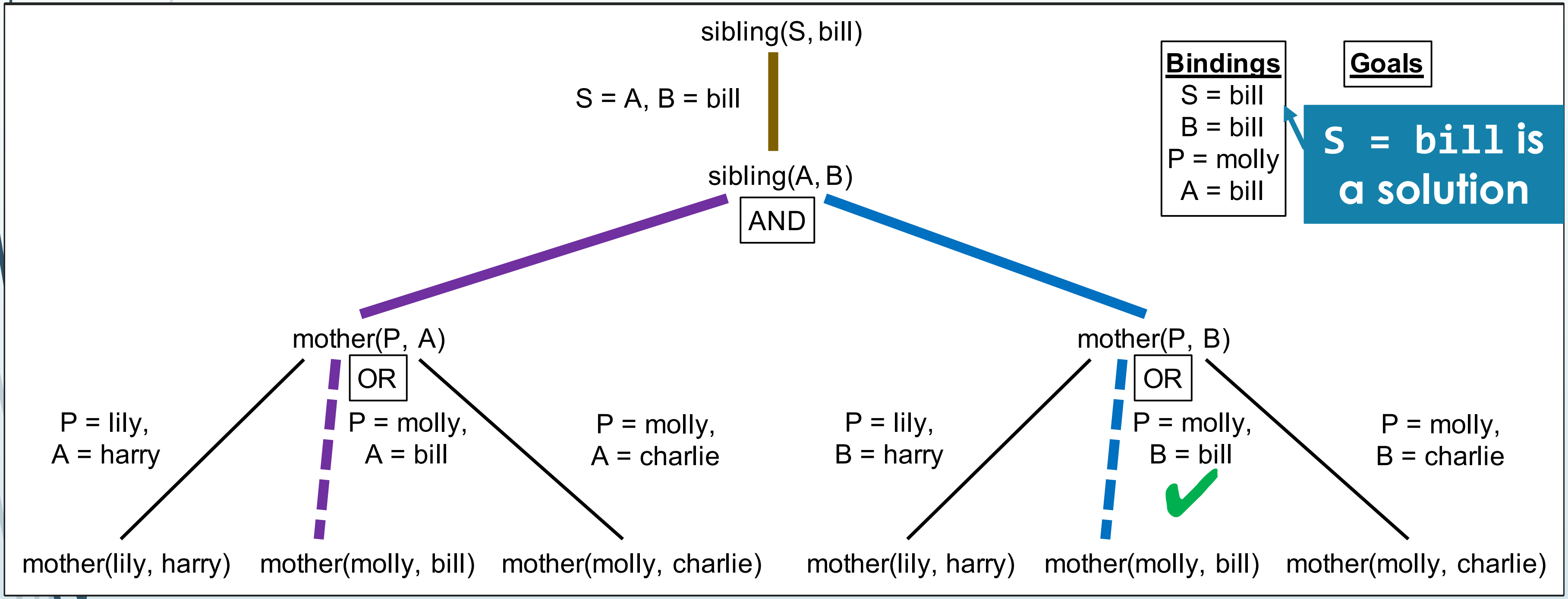
- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



First Solution

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

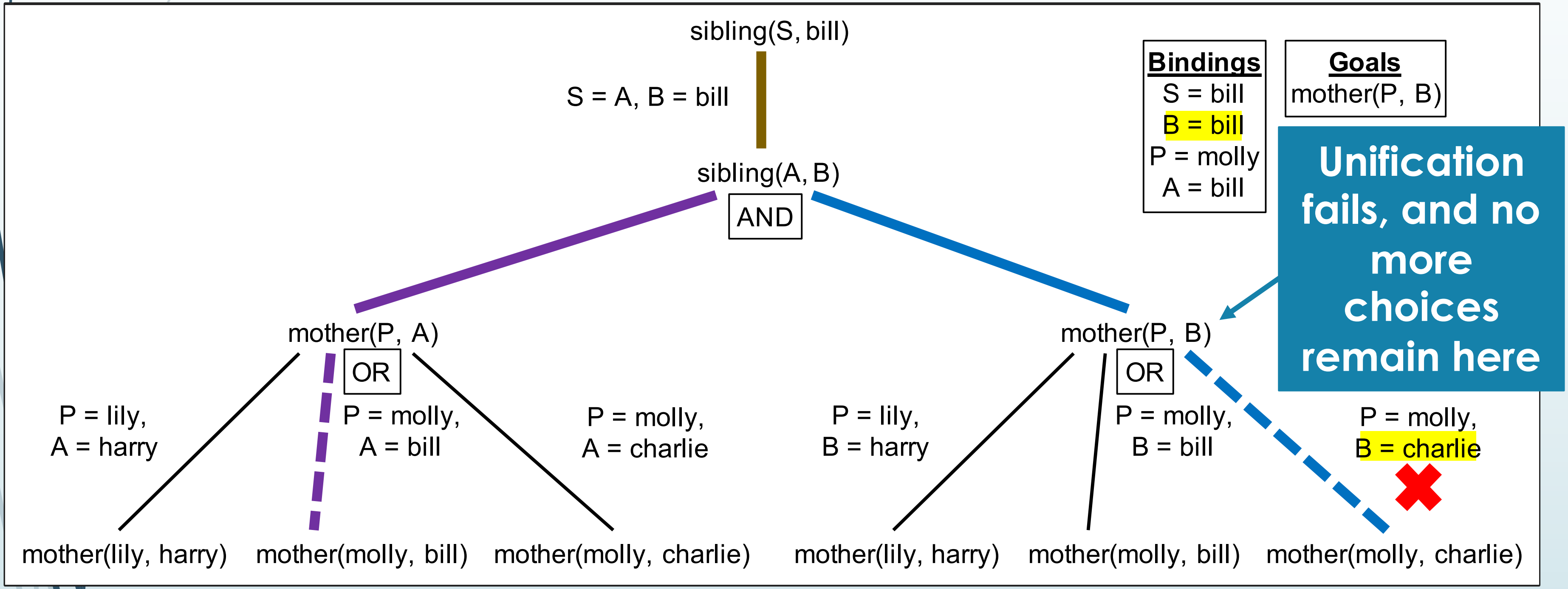
- The search backtracks to the previous choice point, attempting to apply the fact `mother(molly, bill)`
- Unification succeeds, and no goal terms remain



Continuing the Search

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

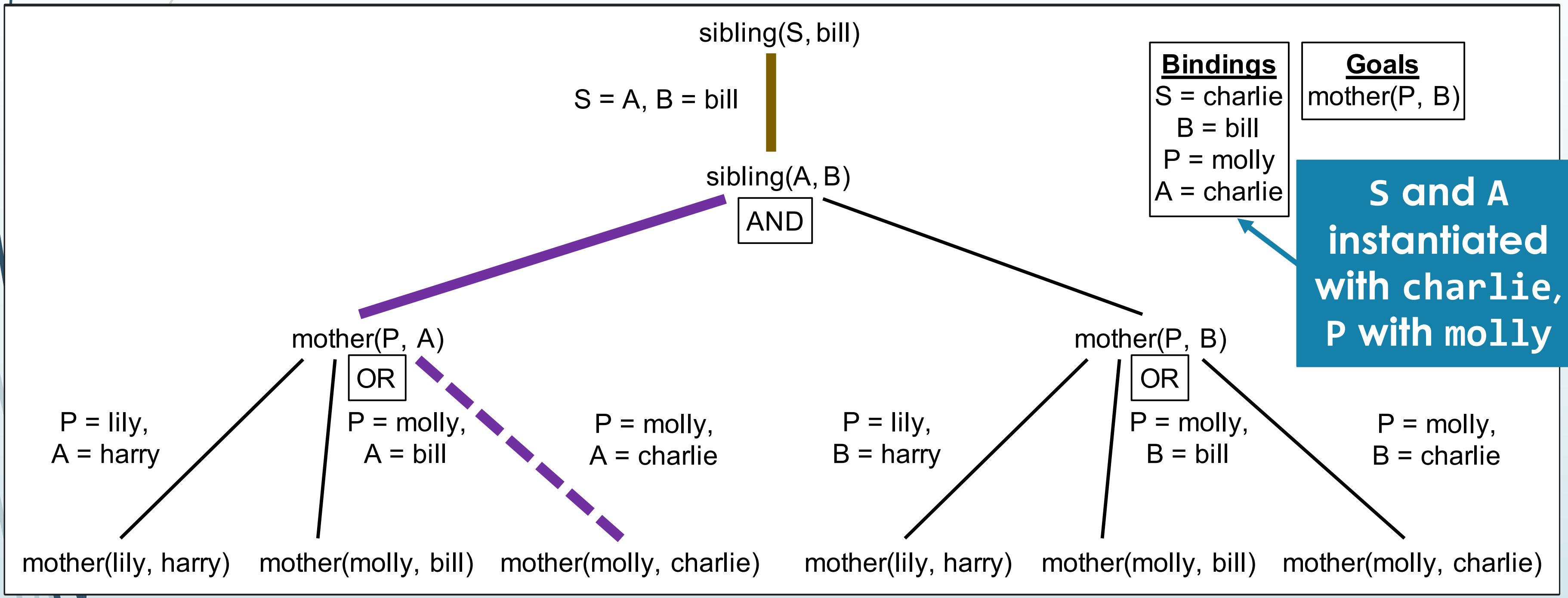
- If we ask the interpreter for another solution, it backtracks to the previous choice point, attempting to apply the fact `mother(molly, charlie)`



Backtracking

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

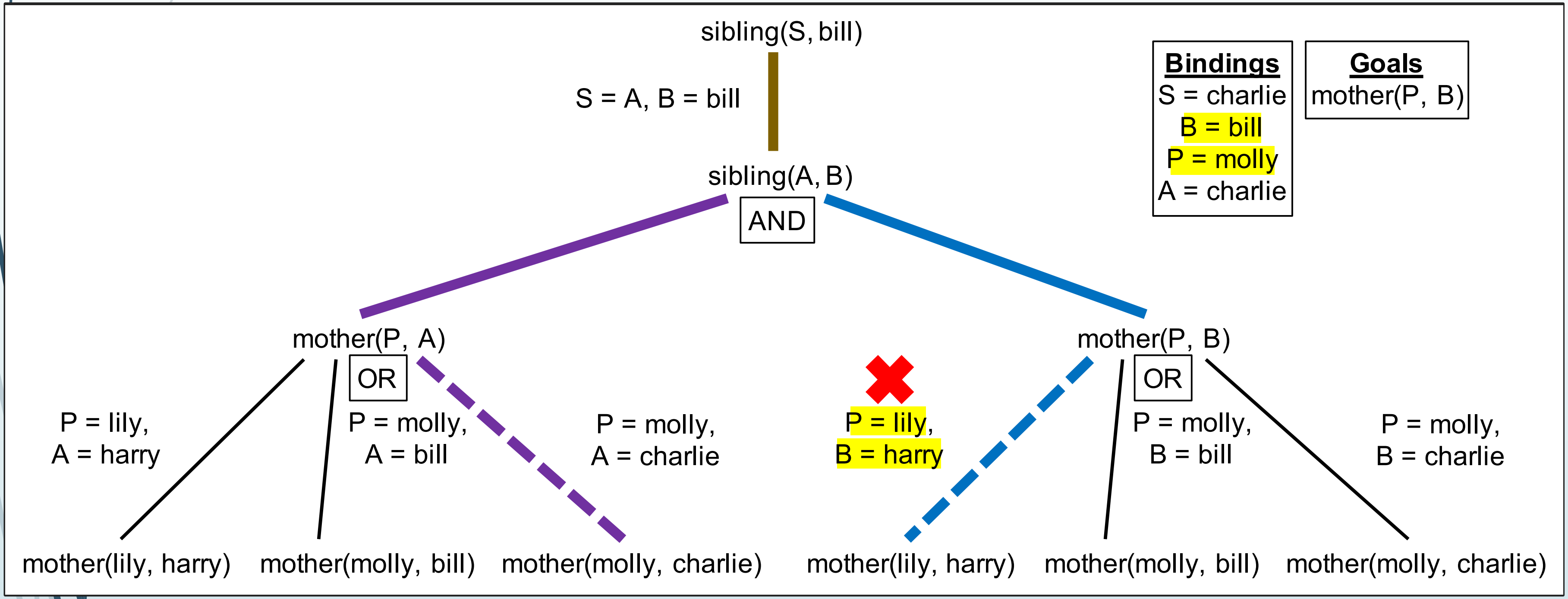
- The search backtracks to the preceding choice point, unifying `mother(P, A)` with `mother(molly, charlie)`



Search Tree

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

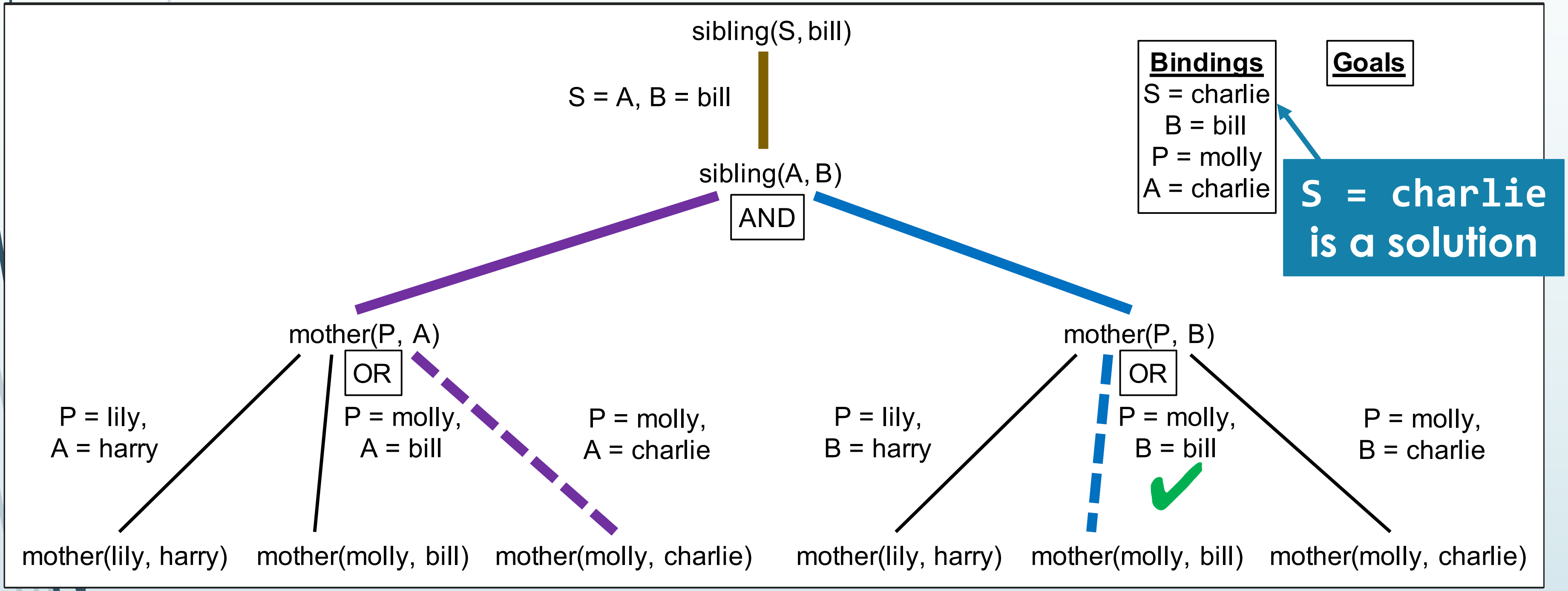
- Then the goal `mother(P, B)` is solved, with an initial choice of applying the fact `mother(lily, harry)`
- However, unification of `B = bill` with `harry` fails



Second Solution

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

- 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

```
sibling(A, B) :-
  mother(P, A), mother(P, B).
mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

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