EECS 490 – Lecture 19

Logic Programming

11/13/17

Announcements

- HW4 due tonight at 8pm
- Project 4 due Tue 11/21 at 8pm

Internal Linkage

- C does not have namespaces, so it uses *linkage* specifiers to avoid name conflicts between translation units
  - Also in C++, since it's mostly compatible with C
- A variable or function at global scope can be declared **`static`**, which specifies *internal linkage*
  - Name will not be visible outside of translation unit, avoiding name conflicts at link stage
- Variables and functions defined, not just declared, in a header should generally be **`static`**

```
static const double PI = 3.1415926535;
static double area(double r) {
  return PI * r * r;
}
```

External Linkage

- A global variable or function has *external linkage* if it does not have the `static` specifier
  - The name will be accessible from other translation units
- An entity with external linkage must have exactly one definition among the translation units of a program
- A function can be declared but not defined by leaving out the function body
- A variable declaration is also a definition, unless it has the extern specifier

```
extern int count;
int count;
```

Just a declaration

A definition that default initializes count

Information Hiding

- Languages often support information hiding at the granularity of a module or package
- In Java, a non-`public` class is available only to the same package
- Java and C# have module or package-level access modifiers for class members
- In C and C++, entities declared with internal linkage in a `.c` or `.cpp` file are not available to other translation units

Opaque Types in C

- In C, struct members can be hidden by providing only a declaration and not the definition of a struct in the header file

```
typedef struct list *stack;
stack stack_make();
void stack_push(stack s, int i);
int stack_top(stack s);
void stack_pop(stack s);
void stack_free(stack s);
```

- Other translation units can make use of the interface, but cannot access members or even directly create an object of an opaque type

```
stack s = stack_make();
stack_push(s, 3);
stack_pop(s);
```

Initialization

- Languages specify semantics for initialization of the contents of a class, module, or package
- In Java, a class is initialized the first time it is used
  - Generally when an instance is created or a static member is accessed for the first time
- In Python, a module's code is executed when it is imported
  - If a module is imported again from the same module, its code does not execute again

Circular Dependencies

- Circular dependencies between modules should be avoided
- Can require restructuring code
- Example:

```
> python3 foo.py
Traceback (most recent call last):
  File "foo.py", line 1, in <module>
    import bar
  File "bar.py", line 1, in <module>
    import foo
  File "foo.py", line 9, in <module>
    print(func1())
  File "foo.py", line 4, in func1
    return bar.func3()
AttributeError: module 'bar' has no
attribute 'func3'
```

foo.py

```
import bar

def func1():
    return bar.func3()

def func2():
    return 2

print(func1())
```

bar.py

```
import foo

def func3():
    return foo.func2()
```

Initialization in C++

- C++ has a multi-step initialization process
  1. **Static initialization**: initialize compile-time constants to their values, and all other variables with static storage duration to zero
  2. **Dynamic initialization**: initialize static-storage variables using their specified initializers
     - Can be delayed until first use of the translation unit
- Within a translation unit, initialization is in program order, with some exceptions
- Order is undefined between translation units
  - Cannot rely on another translation unit being initialized first

Logic Programming

- Imperative programming: express computation as sequences of operations on the program state
- Functional programming: express computation as mappings between function inputs and outputs
- Logic programming: express computation as relations between pieces of data
- First-order predicate calculus is the foundation of logic programming

$$\forall X.\ \exists Y.\ P(X) \vee \neg Q(Y)$$
$$\forall X.\ \exists Y.\ Q(Y) \Rightarrow P(X)$$

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
```

Head of clause — **H**

Body of clause — **B1, B2, ..., BN**

Individual *term* — **BN**

- This is equivalent to

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

with implicit quantifiers.

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:

```
   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

The solution order is deterministic, as we will see shortly.

Break Time!

- We'll start again in five minutes.

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:

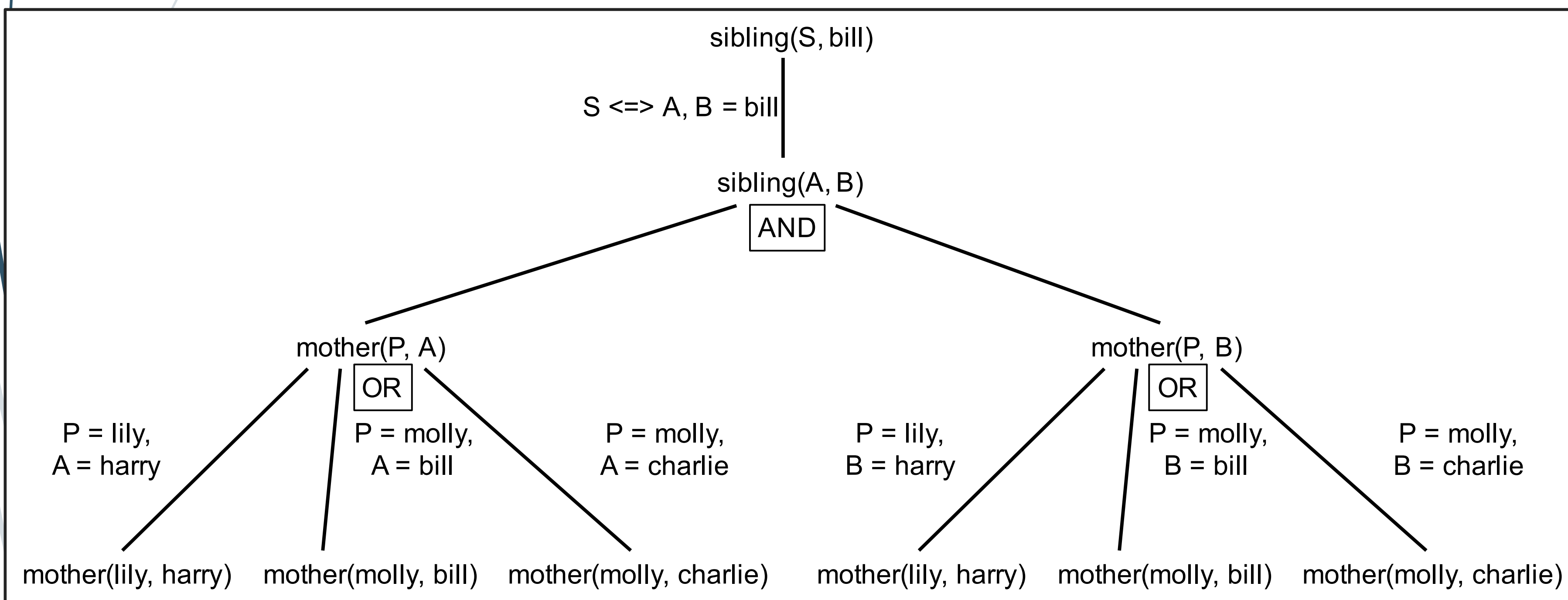
```
sibling(A, B) :- mother(P, A), mother(P, B).

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

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

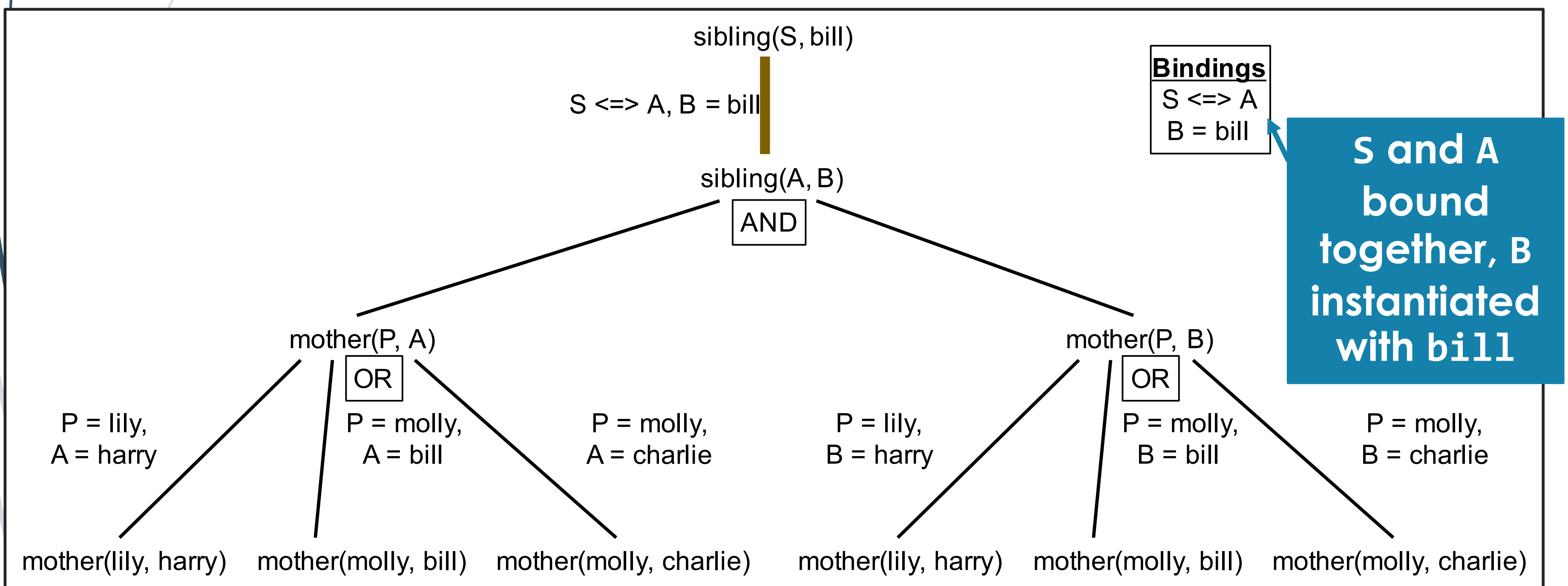
Search Tree

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



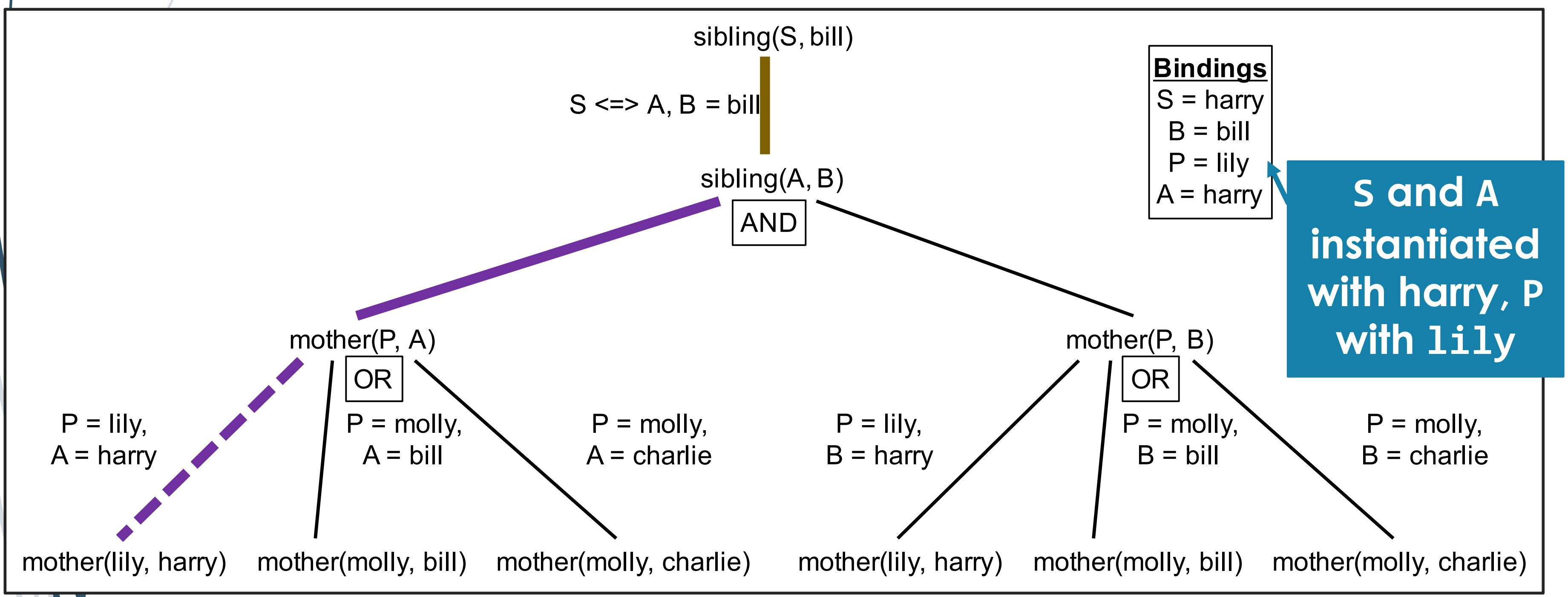
Search Tree

- 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



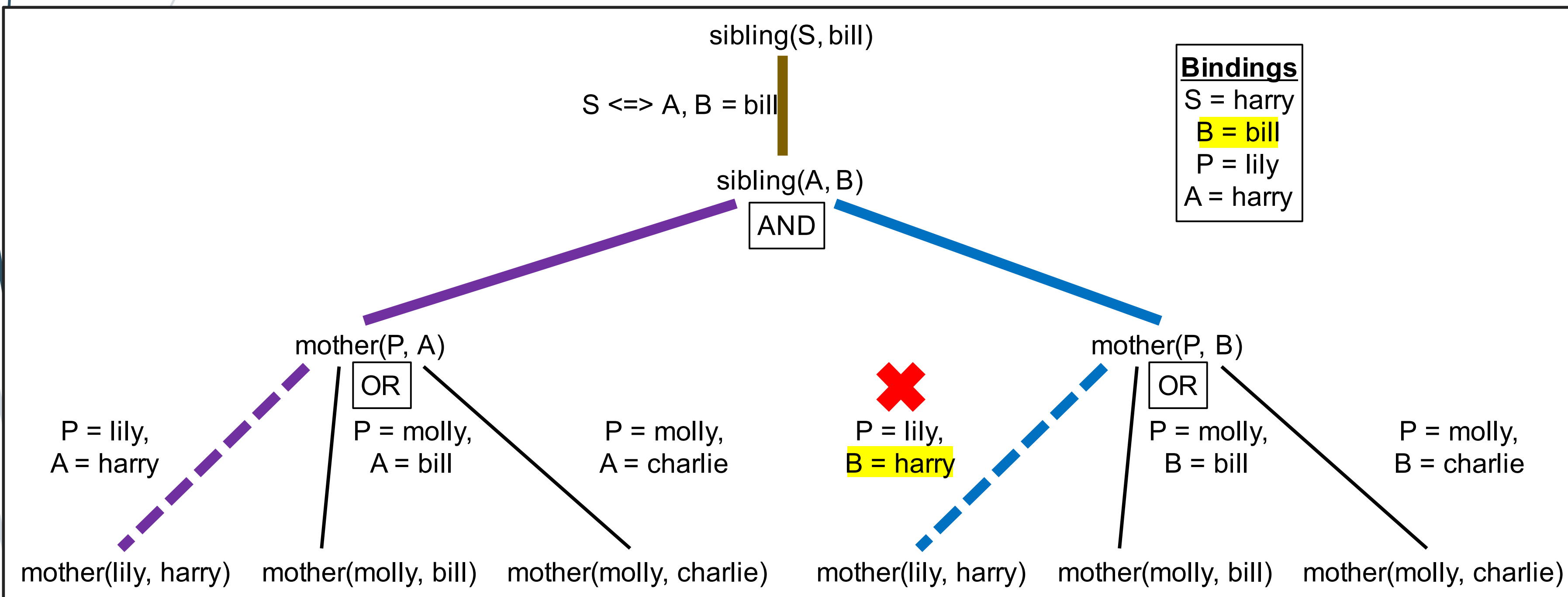
Search Tree

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



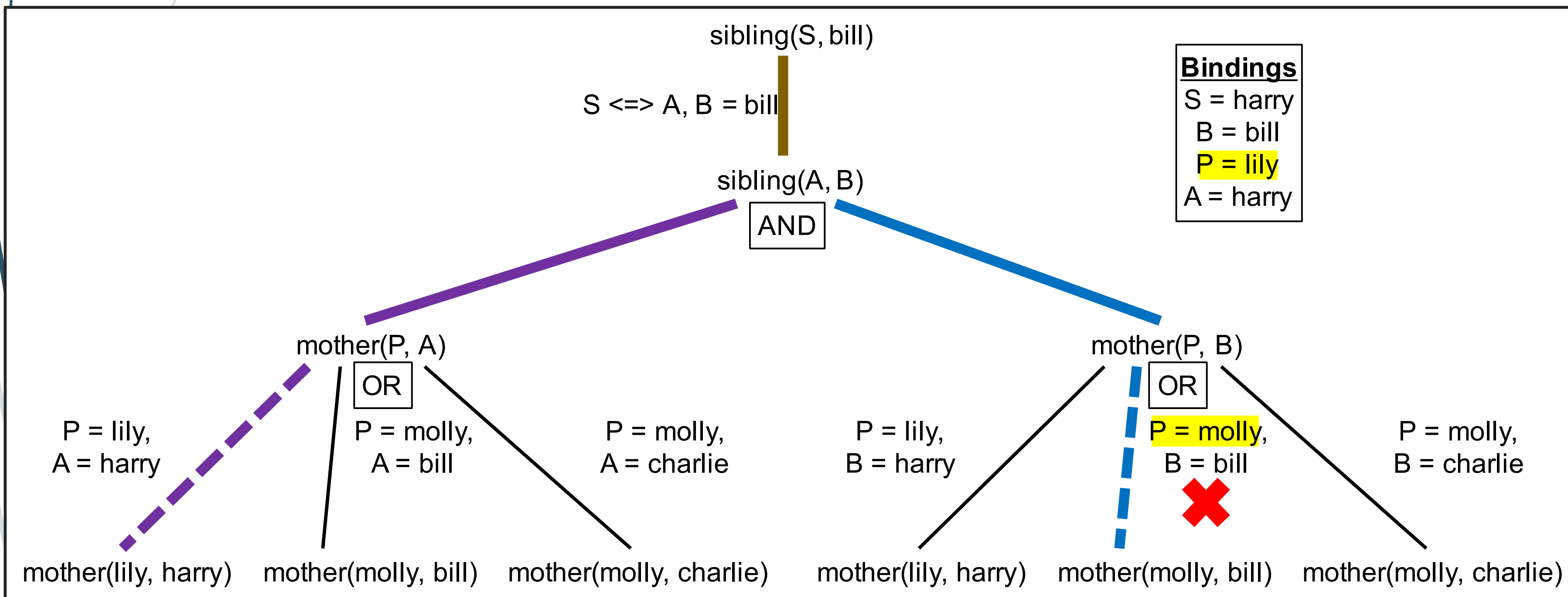
Search Tree

- 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



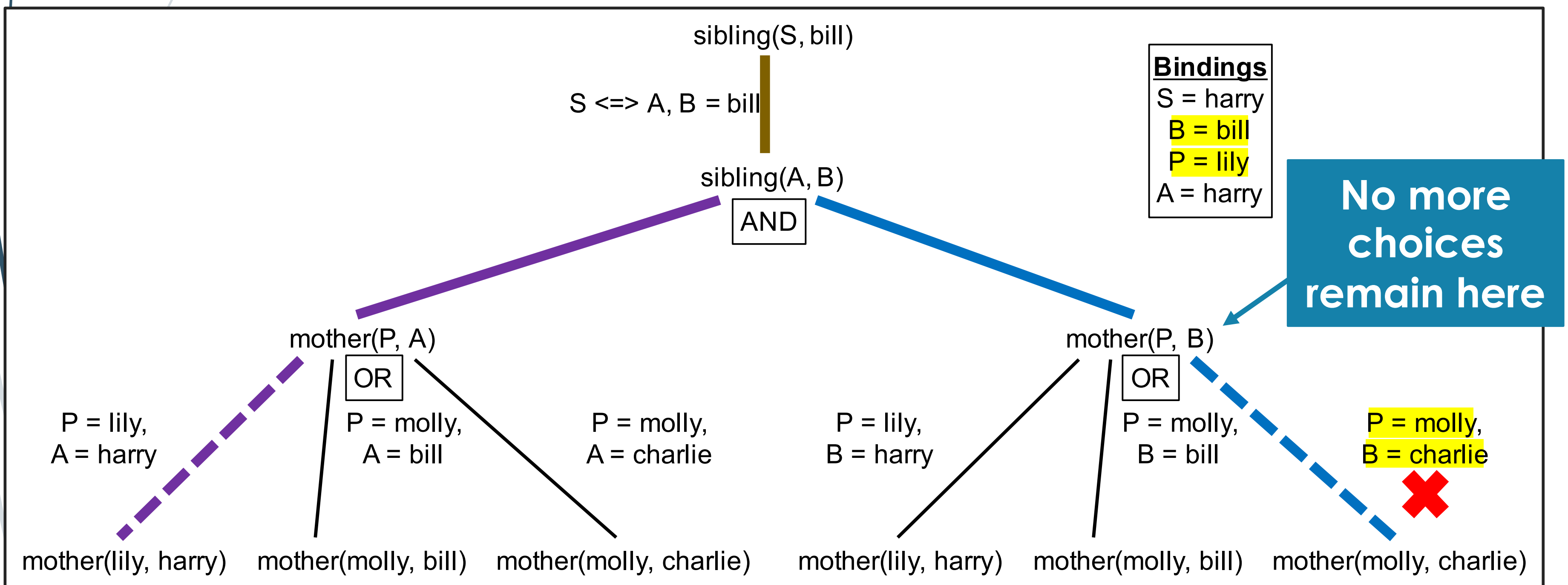
Backtracking

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



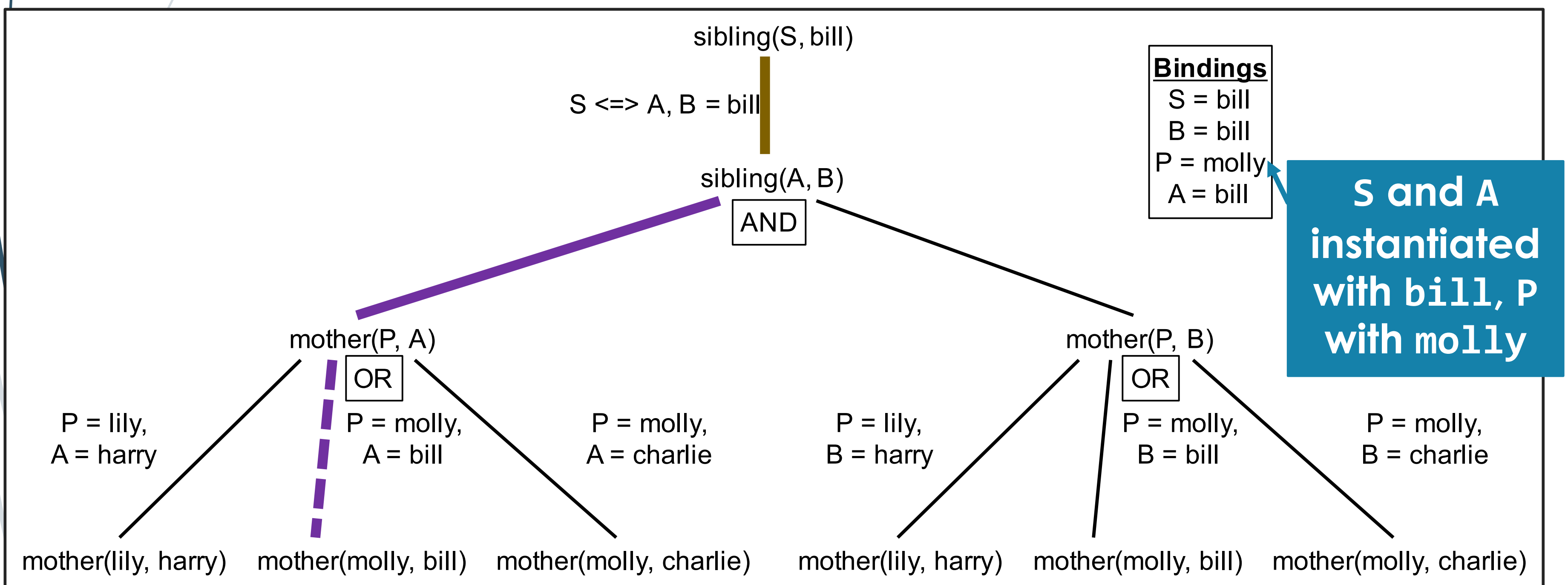
Backtracking

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



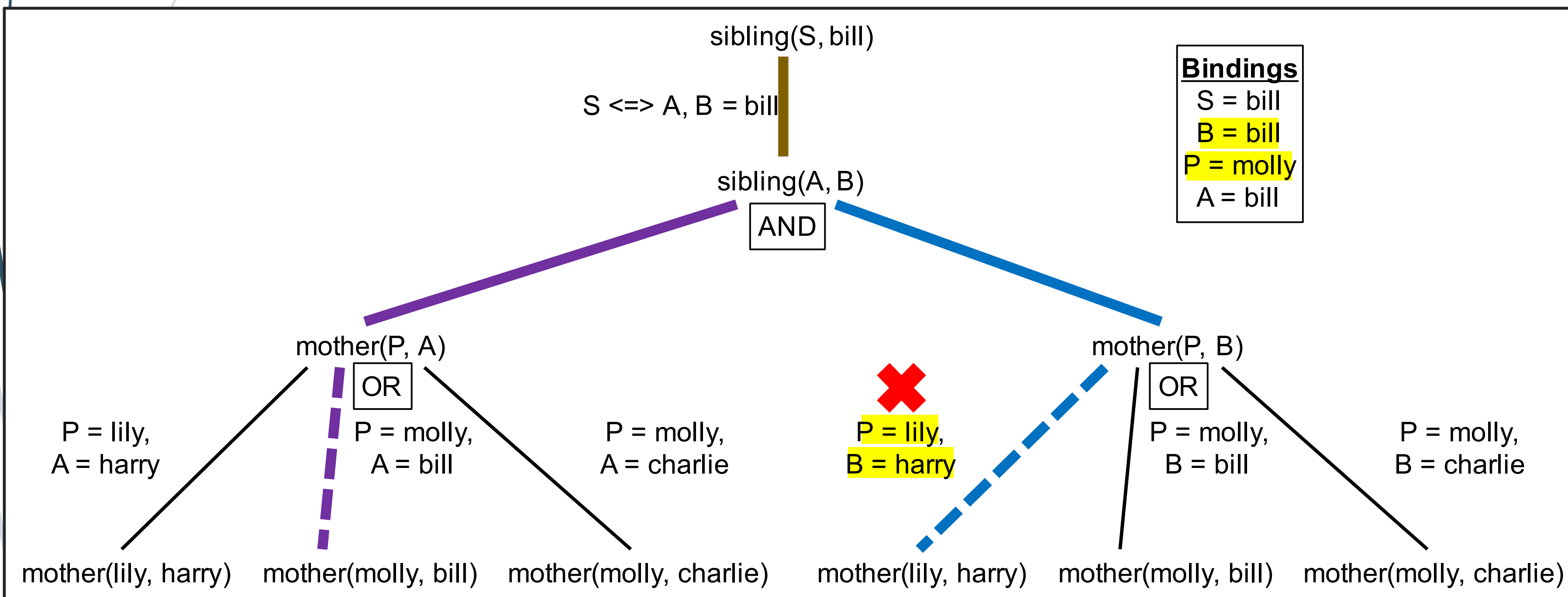
Backtracking

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



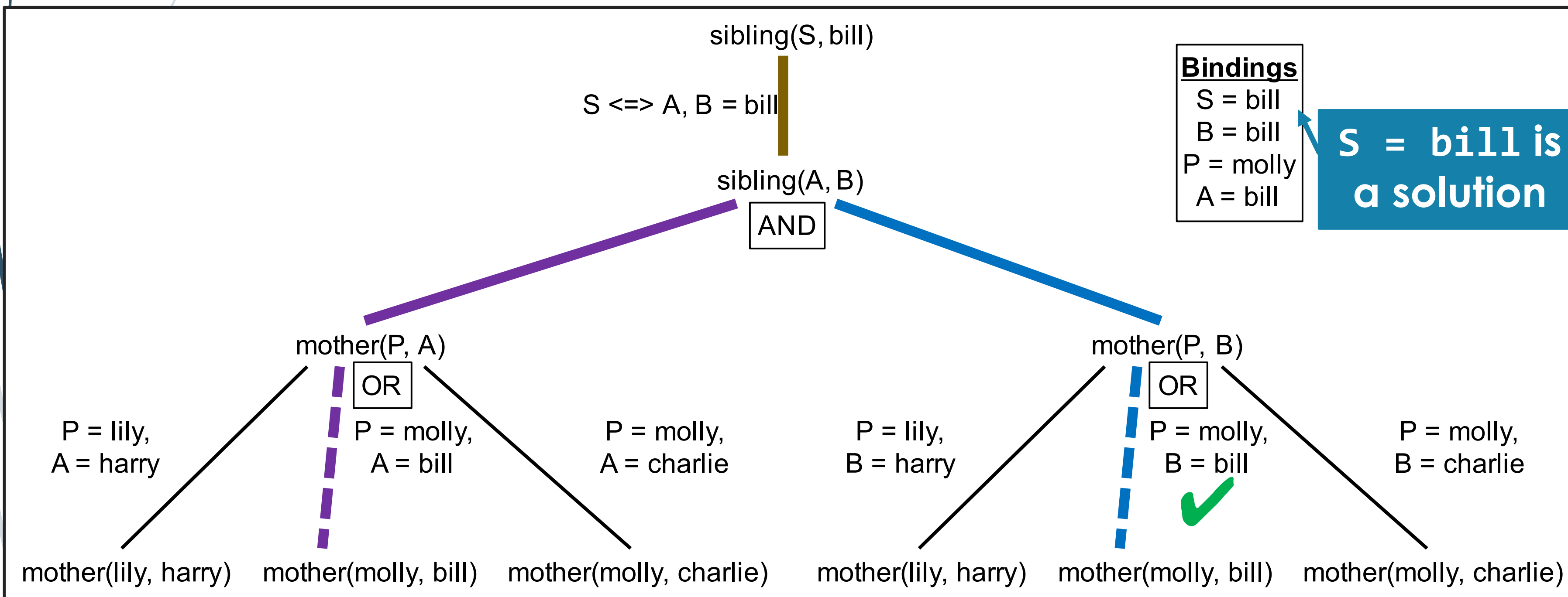
Search Tree

- 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



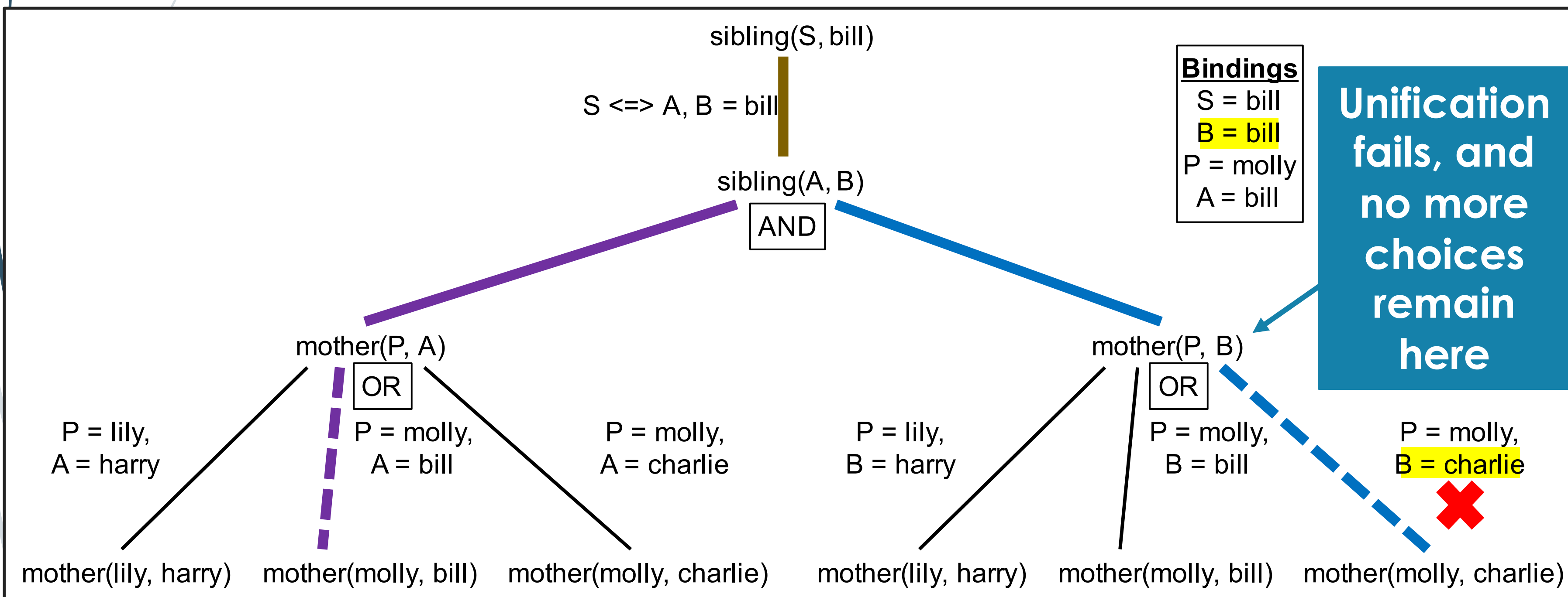
First Solution

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



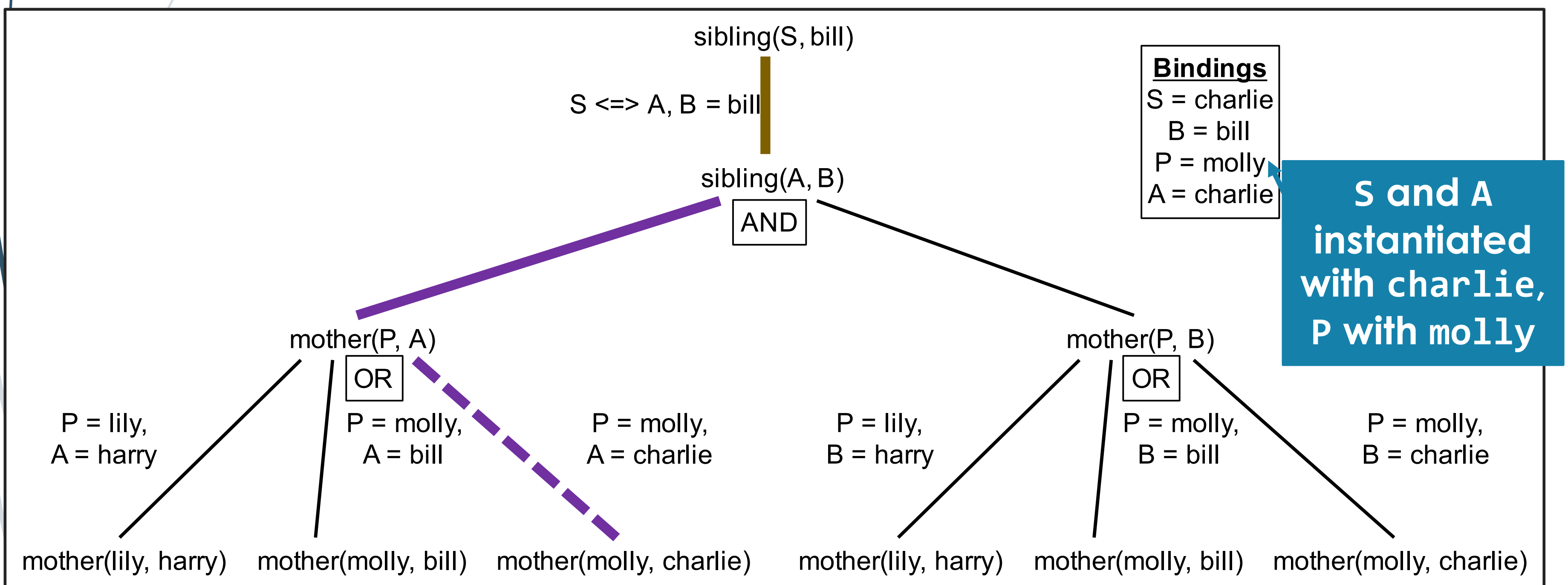
Continuing the Search

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



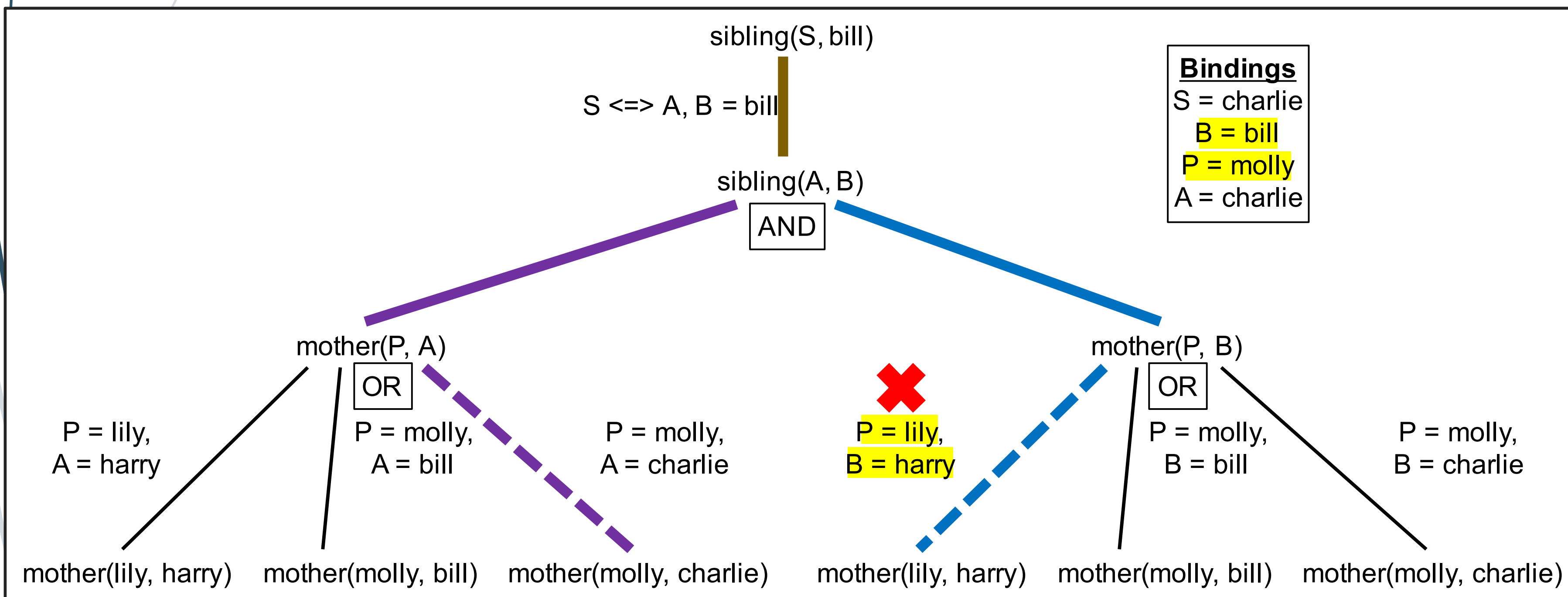
Backtracking

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



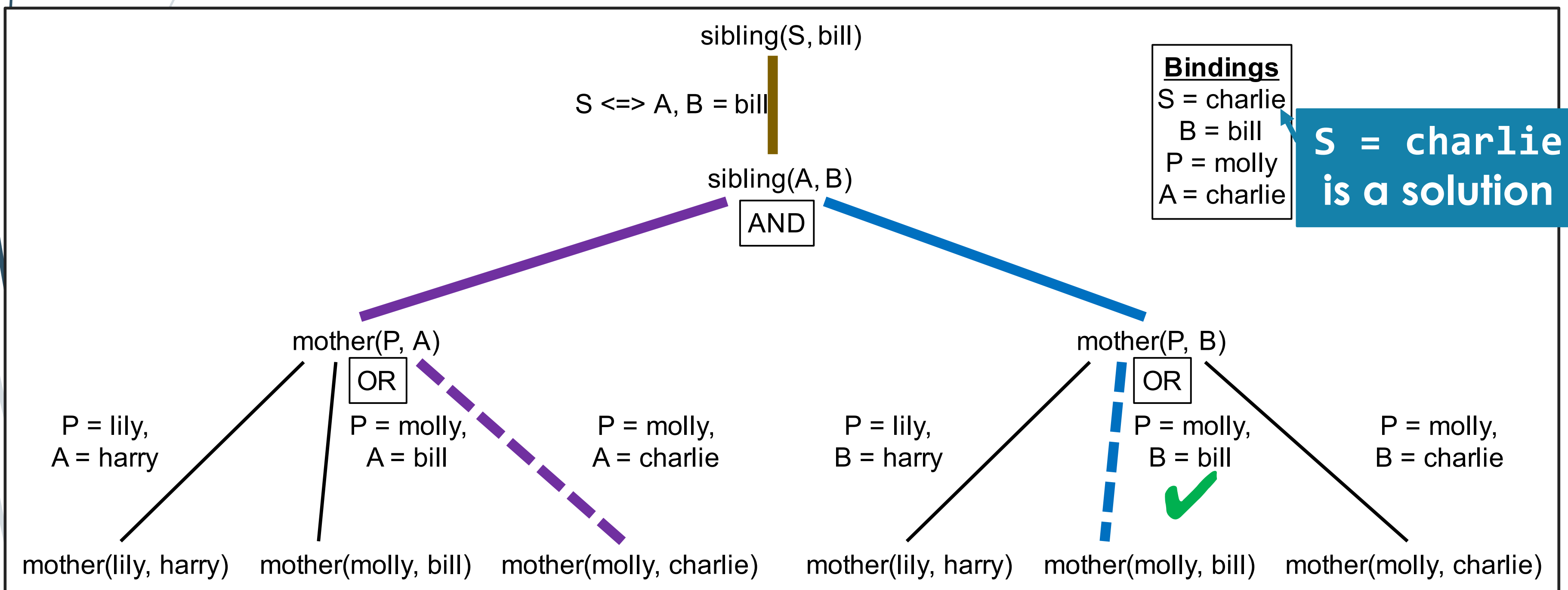
Search Tree

- 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)`