Prolog and Logic Languages

Prolog

- Based on first-order predicate logic
- Original motivation: study of mechanical theorem proving
- Developed in 1970 by Colmerauer & Roussel (Marseilles) and Kowalski (Edinburgh) + others.
- Used in Artificial Intelligence, databases, expert systems.

Prolog Programs

- Program = a bunch of axioms
- Run your program by:
  - Enter a series of facts and declarations
  - Pose a query
  - System tries to prove your query by finding a series of inference steps

- “Philosophically” declarative
- Actual implementations are deterministic

Horn Clauses (Axioms)

- Axioms in logic languages are written:

  $H \leftarrow B1, B2, \ldots., B3$

Facts = clause with head and no body.

Rules = have both head and body.

Query – can be thought of as a clause with no body.

Terms

- H and B are terms.
- Terms =
  - Atoms - begin with lowercase letters: x, y, z, fred
  - Numbers: integers, reals
  - Variables - begin with capital letters: X, Y, Z, Alist
  - Structures: consist of an atom called a functor, and a list of arguments. ex. edge(a,b). line(1,2,4).

Lists

- [] % the empty list
- [1]
- [1,2,3]
- [[1,2], 3] % can be heterogeneous.

The | separates the head and tail of a list:

is [a | [b,c]]

Backward Chaining

START WITH THE GOAL and work backwards, attempting to decompose it into a set of (true) clauses.

This is what the Prolog interpreter does.

Forward Chaining

- START WITH EXISTING FACTS and clauses and work forward, trying to derive the goal.

- Unless the number of facts is very small and the number of rules is large, backward chaining will probably be faster.

Searching the database as a tree

- **DEPTH FIRST** - finds a complete sequence of propositions for the first subgoal before working on the others. (what Prolog uses)

- **BREADTH FIRST** - works on all subgoals in parallel.

- The implementers of Prolog chose depth first because it can be done with a stack (expected to use fewer memory resources than breadth first).

Unification

```
likes(bob,fondue).
likes(sue,fondue).
friends(X,Y) :-
    likes(X, Something),
    likes(Y, Something).
```

% Y is a variable, find out who is friends with Sue.

```
?- friends(sue,Y).
```

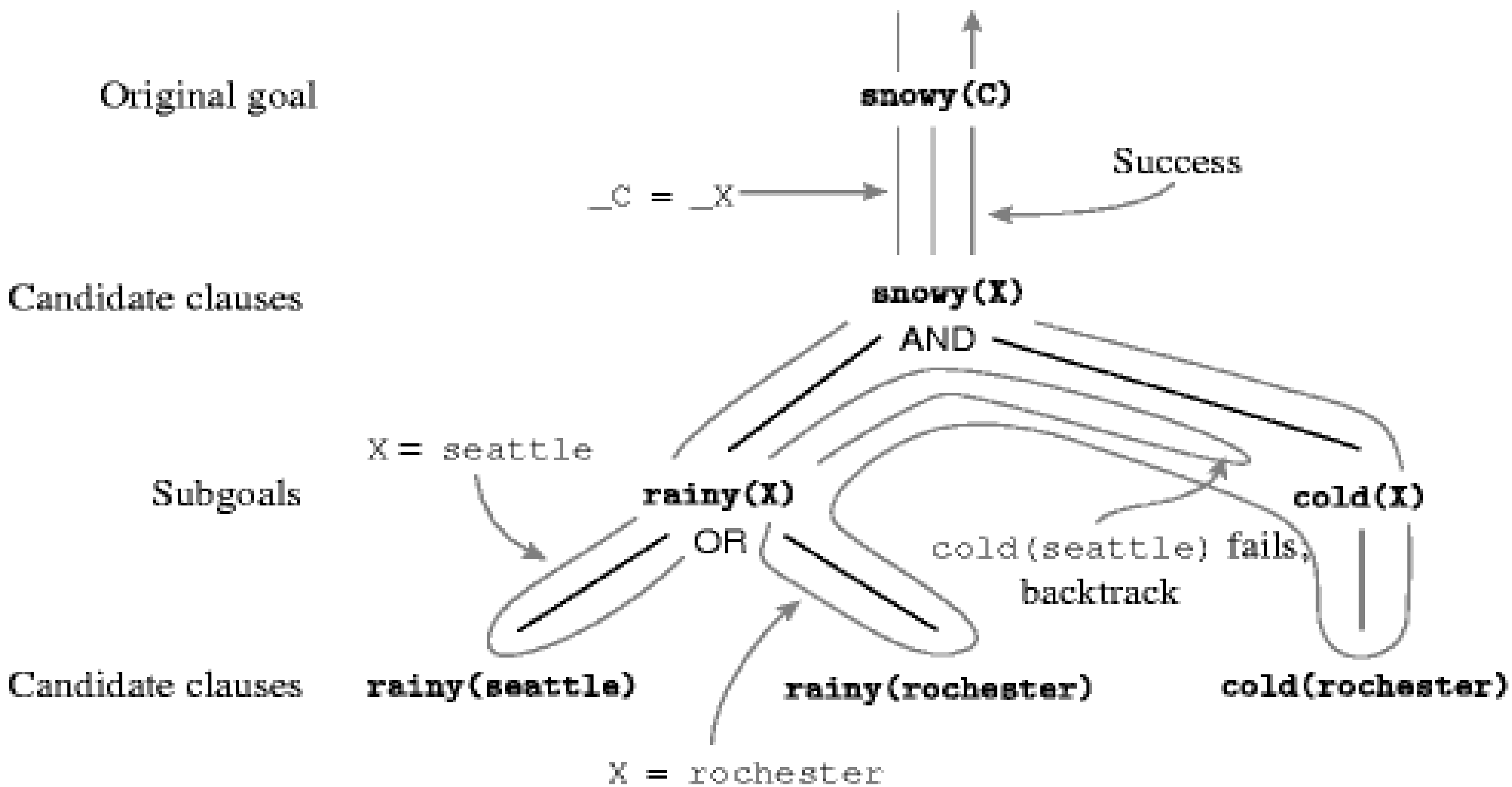
```
friends(sue,Y) :-    % replace X with sue in the clause
    likes(sue, Something),
    likes(Y, Something).
```

We replace the 1st clause in friends with the empty body of the likes(sue,fondue) clause to get:

friends(sue,Y) :-
likes(Y, fondue). %- now we try to satisfy the second goal.
(Finally we will return an answer to the original query like:Y=bob)

Backtracking search

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```



Order of Rule Evaluation

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).
```

- The last two clauses tell us how to determine whether there is a path from node X to node Y.

Order of Rule Evaluation

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- edge(Z, Y), path(X, Z).
```

- The last two clauses tell us how to determine whether there is a path from node X to node Y.
- If we were to reverse the order of the terms on the right-hand side of the final clause?

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- path(X, Z), edge(Z, Y).
```

Order of Rule Evaluation

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, X).
path(X, Y) :- path(X, Z), edge(Z, Y).
```

- The Prolog interpreter would search for a node Z that is reachable from X before checking to see whether there is an edge from Z to Y.
- The program would still work, but it would not be as efficient.

- What would happen if in addition we were to reverse the order the last two clauses?

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```

Improperly Ordered Declarations

```
edge(a, b). edge(b, c). edge(c, d).
edge(d, e). edge(b, e). edge(d, f).
path(X, Y) :- path(X, Z), edge(Z, Y).
path(X, X).
```

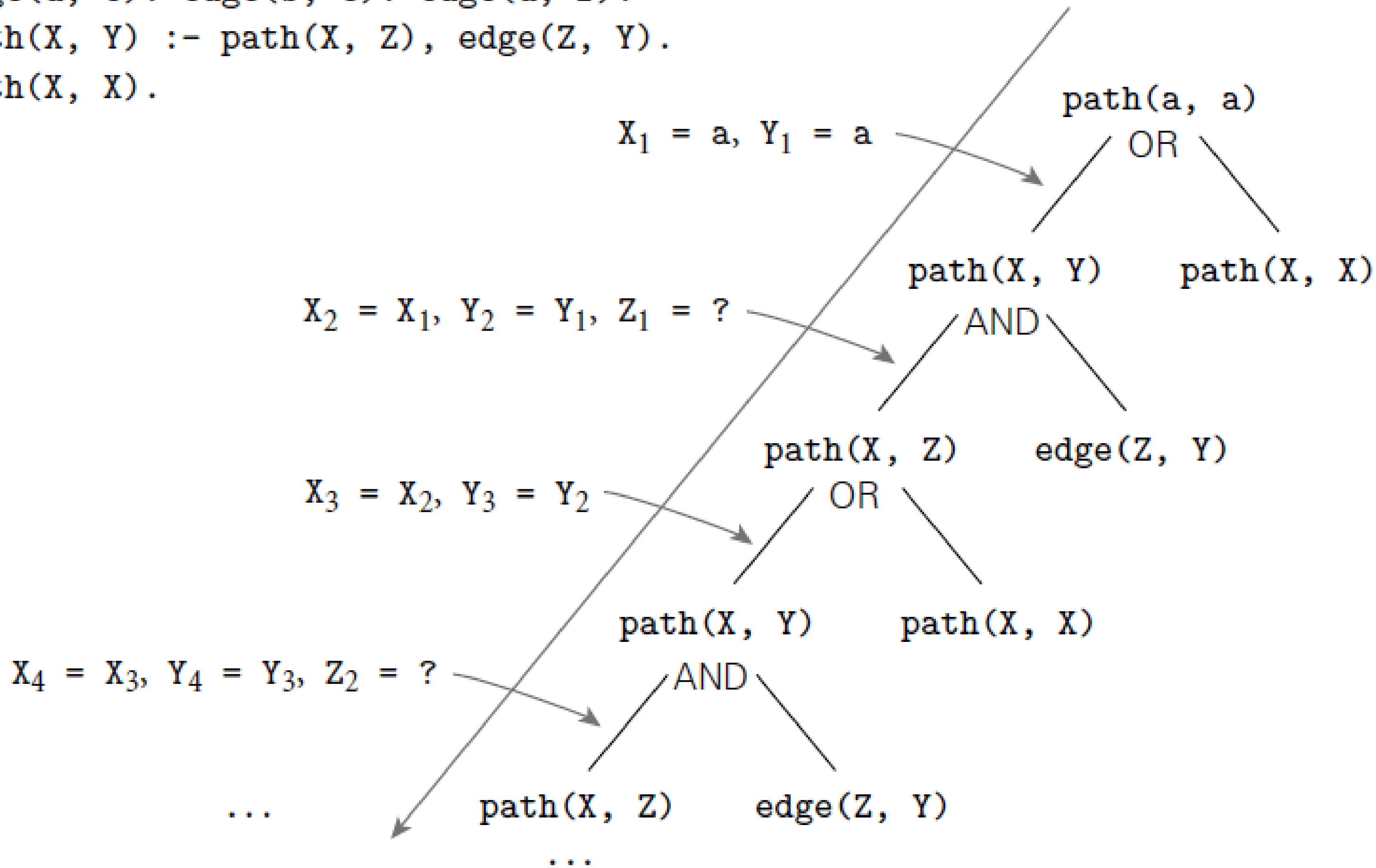


Figure 12.2 Infinite regression in Prolog. In this figure even a simple query like `?- path(a, a)` will never terminate: the interpreter will never find the trivial branch.

Backtracking

- Consider a piece-wise function
  - if x < 3, then y = 0
  - if x >=3 and x < 6, then y = 2
  - if x >= 6, then y = 4
- Let's encode this in Prolog

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

Backtracking

- Let's encode this in Prolog

```
f(X,0) :- X < 3.
f(X,2) :- 3 =< X, X < 6.
f(X,4) :- 6 =< X.
```

- Consider

```
?- f(1,Y), 2<Y.
```

- This matches the f(X,0) predicate, which succeeds
  – Y is then instantiated to 0
  – The second part (2<Y) causes this query to fail
- Prolog then backtracks and tries the other predicates
  – But if the first one succeeds, the others will always fail!
  – This, the extra backtracking is unnecessary

Backtracking

- We want to tell Prolog that if the first one succeeds, there is no need to try the others

- We do this with a cut:

```
f(X,0) :- X<3, !.
f(X,2) :- 3 =< X, X<6, !.
f(X,4) :- 6 =< X.
```

- The cut ('!') prevents Prolog from backtracking backwards through the cut

Backtracking

- New Prolog code:

```
f(X,0) :- X<3, !.
f(X,2) :- 3 =< X, X<6, !.
f(X,4) :- 6 =< X.
```

- Note that if the first predicate fails, we know that x >= 3
  - Thus, we don't have to check it in the second one.
  - Similarly with x>=6 for the second and third predicates

- Revised Prolog code:

```
f(X,0) :- X<3, !.
f(X,2) :- X<6, !.
f(X,4).
```

Backtracking

- What if we removed the cuts:

```
f(X,0) :- X<3.
f(X,2) :- X<6.
f(X,4).
```

- Then the following query:

```
?- f(1,X).
```

- Will produce three answers (0, 2, 4)

Example

- Maximum of two values without a cut:

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

- Maximum of two values with a cut:

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

Example

- Exercise 1.3.

References

- M. L. Scott, Programming Language Pragmatics (4th Edition), Morgan Kaufmann, 2016.
- Lecture notes of Aaron Bloomfield, CS 415.