Logic Programming:
Terms, unification and proof search

Alan Smaill

Sep 23 2013

School of **informatics**

- Compound terms
- Equality and unification
- How Prolog searches for answers

So far we have seen . . .

▶ Atoms: `homer marge 'Mr. Burns'`

▶ Variables: `X Y Z MR_BURNS`

We also have . . .

▶ Numbers: `1 2 3 42 -0.12435`

▶ **Complex terms**

▶ Additional **constants** and **infix operators**

▶ A complex term is of the form

$$f(t_1, \ldots, t_n)$$

▶ where f is an atom and $t_1, \ldots, t_n$ are terms

▶ A complex term is of the form

$$f(t_1, \ldots, t_n)$$

▶ where f is an atom and $t_1, \ldots, t_n$ are terms

Examples:

```
f(1,2)     node(leaf,leaf)     cons(42,nil)
household(homer, marge, bart, lisa, maggie)
```

Lists are built-in (and very useful) data structures

Syntax:

```
[1,2,3,4]
[a,[1,2,3],42,'forty-two']
[a,b,c|Xs]
```

Lots more on this next week ...

Prolog has built-in **constants** and **infix operators**.

Examples:

- ▸ Equality: t = u (or =(t,u))
- ▸ Pairing: (t,u) (or ,(t,u))
- ▸ Empty list: []
- ▸ *cons* – list from first element and rest: [X|Y] (or .(X,Y))

You can also define your own infix operators!

The equation `t = u` is a basic goal
with a special meaning

What happens if we ask:

```
?- X = c.
?- f(X,g(Y,Z)) = f(c,g(X,Y)).
?- f(X,g(Y,f(X))) = f(c,g(X,Y)).
```

And how does it do that?

```
?- X = c.
X=c
yes

?- f(X,g(Y,Z)) = f(c,g(X,Y)).
X=c
Y=c
Z=c
yes

?- f(X,g(Y,f(X))) = f(c,g(X,Y)).
no
```
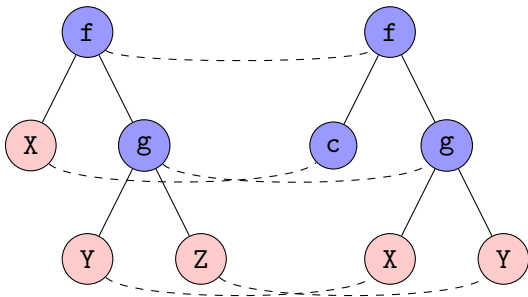
- A **substitution** is a mapping from variables to terms
  - $X_1 = t_1, \ldots, X_n = t_n$

- Given two terms t and u
  - with free variables $X_1, \ldots, X_n$,

- a unifier is a substitution that makes t and u identical when applied to t and u.

Example I

School of **informatics**

$$f(X,g(Y,Z)) = f(c,g(X,Y))$$

X=c
Y=X
Z=Y

$$f(X,g(Y,Z)) = f(c,g(X,Y))$$

X=c
Y=c
Z=c
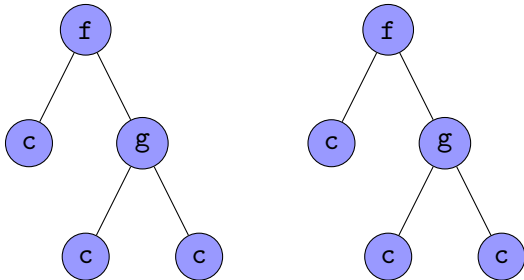
Example II

f(X,g(Y,f(X))) = f(c,g(X,Y))

X=c
Y=X
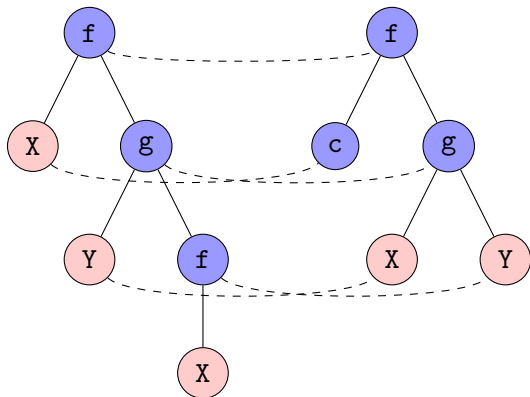
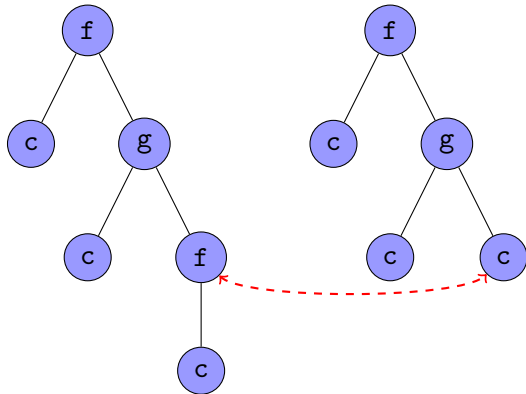`f(X,g(Y,f(X))) = f(c,g(X,Y))`

X=c
Y=c
Y=f(X)
f(X)=c???

▶ Consider a general unification problem

$$t_1 = u_1, t_2 = u_2, \ldots, t_n = u_n$$

▸ Consider a general unification problem

$$t_1 = u_1, t_2 = u_2, \ldots, t_n = u_n$$

▸ Reduce the problem by decomposing each equation into one or more "smaller" equations

▸ Succeed if we reduce to a "solved form", otherwise fail.

▶ Two function applications unify if the head symbols are equal, and the corresponding arguments unify:

$$f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n), \quad P \quad \Rightarrow$$
$$t_1 = u_1, \ \ldots, \ t_n = u_n, \quad P$$

School of **informatics**

- Two function applications unify if the head symbols are equal, and the corresponding arguments unify:

$$f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n), \quad P \quad \Rightarrow$$
$$t_1 = u_1, \; \ldots, \; t_n = u_n, \quad P$$

- Must have same name, and equal number of arguments:

$$f(\ldots) = c, \quad P \quad \Rightarrow \quad \textit{fail}$$
$$f(\ldots) = g(\ldots), \quad P \quad \Rightarrow \quad \textit{fail}$$

▶ Otherwise, a variable X unifies with a term t, provided X does not occur in t:

▶ proceed by substituting t for X in P:

$$X = t, \quad P \quad \Rightarrow \quad P[t/X]$$

occurs check: provided X does not occur in t

▶ What happens if we try to unify *X* with something that *contains X*?

```
?- X = f(X).
```

▶ What happens if we try to unify X with something that *contains X*?

```
?- X = f(X).
```

▶ Logically this should fail
  there is no (finite) unifier!

▶ Most Prolog implementations skip this check for efficiency reasons
  ▶ can use `unify_with_occurs_check/2`

School of
informatics

The query is run by trying to find a solution to the goal using the clauses:

- ▶ Unification is used to match goals and clauses
- ▶ There may be zero, one, or many solutions
- ▶ Execution may backtrack

The formal model is called **SLD** resolution, which you'll see in the theory lectures

Depth-first search

Basic Idea:

To solve atomic goal A:

▶ **If** B is a fact in the program, and $\theta(A) = \theta(B)$, then return answer θ;

▶ **else**,

   if $B \text{ :- } G_1, ..., G_n$ is a clause in the program,
   and θ unifies A with B,
      then solve $\theta(G_1), \ldots, \theta(G_n)$

▶ **else** give up on this goal:

   ▶ **backtrack** to last choice point

School of
informatics

Basic Idea:

To solve atomic goal A:

▶ **If** B is a fact in the program, and $\theta(A) = \theta(B)$, then return answer θ;

▶ **else**,

    if $B :\text{-} G_1, ..., G_n$ is a clause in the program,
    and θ unifies A with B,
        then solve $\theta(G_1), \ldots, \theta(G_n)$

▶ **else** give up on this goal:

    ▶ **backtrack** to last choice point

▶ Clauses are tried in **declaration order**

▶ Compound goals are tried in **left-right order**

Prolog tries clauses in order of appearance in the program.
We look at a couple of **search trees** for query execution.
Assume: foo(a). foo(b). foo(c).
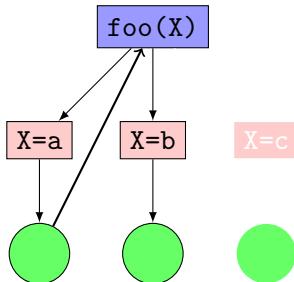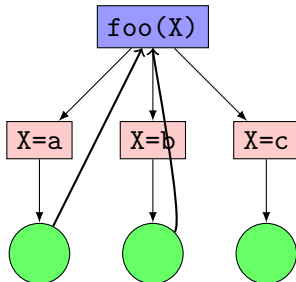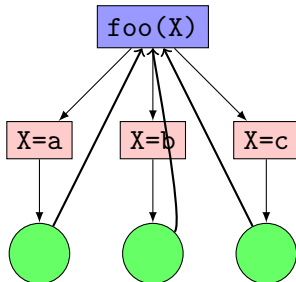then:

?- foo(X).

foo(X)

X=a   X=b   X=c

Prolog tries clauses in order of appearance in the program.
We look at a couple of **search trees** for query execution.
Assume: `foo(a).  foo(b).  foo(c).`
then:

```
?- foo(X).
X=a
```
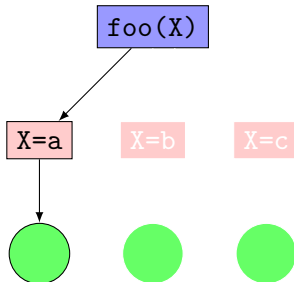
Prolog tries clauses in order of appearance in the program.
We look at a couple of **search trees** for query execution.
Assume: `foo(a).  foo(b).  foo(c).`
then:

```
?- foo(X).
X=a ;
X=b
```

Prolog tries clauses in order of appearance in the program.
We look at a couple of **search trees** for query execution.
Assume: `foo(a).  foo(b).  foo(c).`
then:

```
?- foo(X).
X=a ;
X=b ;
X=c
```

Prolog tries clauses in order of appearance in the program.
We look at a couple of **search trees** for query execution.
Assume: foo(a). foo(b). foo(c).
then:

```
?- foo(X).
X=a ;
X=b ;
X=c ;
no
```

Prolog *backtracks* to the last choice point if a sub-goal fails.
Assume: bar(b).   bar(c).   baz(c). then:

```
?- bar(X),baz(X).
```

bar(X),baz(X)

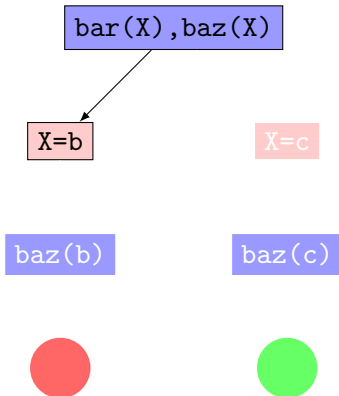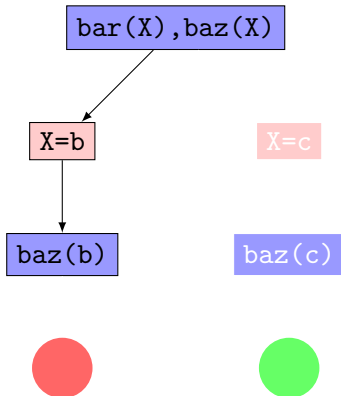X=b   X=c
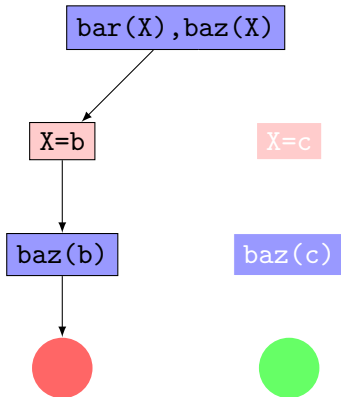
baz(b)   baz(c)

Prolog *backtracks* to the last choice point if a sub-goal fails.
Assume: `bar(b).  bar(c).  baz(c).` then:

```
?- bar(X),baz(X).
```

Prolog *backtracks* to the last choice point if a sub-goal fails.
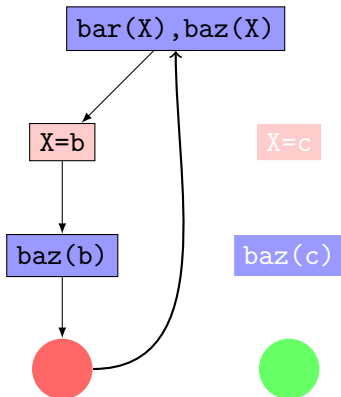Assume: `bar(b). bar(c). baz(c).` then:

`?- bar(X),baz(X).`

informatics School of

Prolog *backtracks* to the last choice point if a sub-goal fails.
Assume: `bar(b). bar(c). baz(c).` then:

```
?- bar(X),baz(X).
```

Prolog *backtracks* to the last choice point if a sub-goal fails.
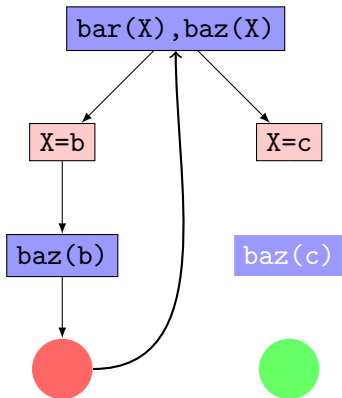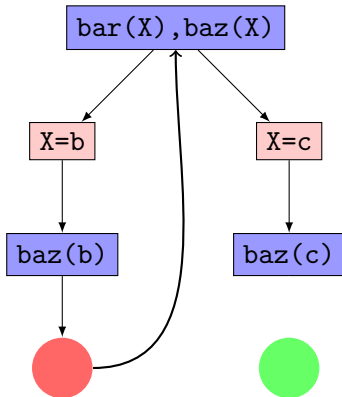Assume: `bar(b). bar(c). baz(c).` then:

`?- bar(X),baz(X).`

informatics

Prolog *backtracks* to the last choice point if a sub-goal fails.
Assume: `bar(b). bar(c). baz(c).` then:
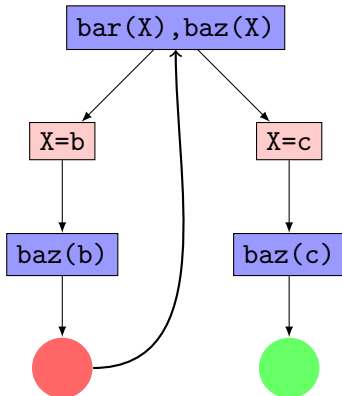
`?- bar(X),baz(X).`

Prolog *backtracks* to the last choice point if a sub-goal fails.
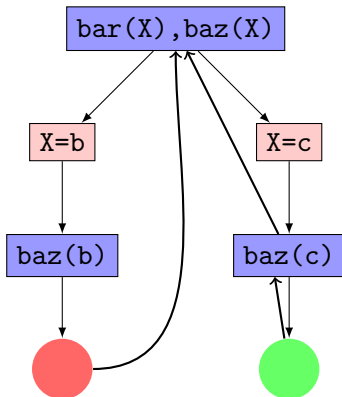Assume: `bar(b).` `bar(c).` `baz(c).` then:

`?- bar(X),baz(X).`

Prolog *backtracks* to the last choice point if a sub-goal fails.
Assume: bar(b). bar(c). baz(c). then:

```
?- bar(X),baz(X).
X = c
```

Prolog *backtracks* to the last choice point if a sub-goal fails.
Assume: `bar(b).  bar(c).  baz(c).` then:

```
?- bar(X),baz(X).
X = c ;
no
```

informatics

▶ Common Prolog programming idiom:

```
find(X) :- generate(X), test(X).
```

where:

  ▶ `generate(X)` produces candidates on backtracking
  ▶ `test(X)` succeeds or fails on candidates

▶ Common Prolog programming idiom:

```
find(X) :- generate(X), test(X).
```

where:

  ▶ `generate(X)` produces candidates on backtracking
  ▶ `test(X)` succeeds or fails on candidates

▶ Use this to constrain (maybe infinite) search spaces;

▶ Can use different generators to get different search strategies besides depth-first.

▶ Recursion
▶ Lists
▶ Trees, data structures

For further reading, see LPN ch. 2.