Declarative and Logic Programming

Mooly Sagiv
Adapted from Peter Hawkins
Jeff Ullman (Stanford)
Why declarative programming

- Turing complete languages (C, C++, Java, Javascript, Haskel, ...) are powerful
  - But require a lot of efforts to do simple things
  - Especially for non-expert programmers
  - Error-prone
  - Hard to be optimize

- Declarative programming provides an alternative
  - Restrict expressive power
  - High level abstractions to perform common tasks
  - Interpreter/Compiler guarantees efficiency
  - Useful to demonstrate some of the concepts in the course
Where can we fly from Denver?

SQL: SELECT dest FROM flight WHERE source="den";
Flight database SQL vs. Datalog

Where can we fly from Denver?

SQL: SELECT dest FROM flight WHERE source="den";

Datalog: flight(den, X).

X = chi;
X = dal;
no

X is a logical variable

Instantiated with the satisfying assignments
Which intermediate airport can I go between San Francisco and Chicago?

Datalog: flight(sfo, X), flight(X, chi).

X = den;
X = dal;
no
Recursive queries Datalog

reachable(sfo, X).
reachable(sfo, Y).
reachable(sfo, Y) :- reachable(sfo, Z), flight(Z, Y).

Where can we reach from San Francisco?

reachable(sfo, X).
X=den;
X=dal;
X=chi;
no
Why Not Just Use SQL?

1. Recursion is much easier to express in Datalog
2. Rules express things that go on in both FROM and WHERE clauses, and let us state some general principles
Syntax of Datalog

Datalog rule syntax:

\[ \text{<result>} :\text{- condition1, condition2, \ldots, conditionN}. \]

- Body consists of one or more conditions (input tables)
- Head is an output table
  - Recursive rules: result of head in rule body
Terminology and Convention

reachable(S,D) :- flight(S,Z), reachable(Z,D).

• An atom is a predicate, or relation name with arguments
• Convention: Variables begin with a capital, predicates begin with lower-case.
• The head is an atom; the body is the AND of one or more atoms.
• Extensional database predicates (EDB) – source tables.
• Intensional database predicates (IDB) – derived tables.
Datalog Programs

• A collection of rules is a (Datalog) program
  – *Order of rules and conjuncts is immaterial*

• Each program has a distinguished IDB predicate that represents the result of the program

• Sometimes given query
  – \text{flight}(sfo, X), \text{flight}(X, \text{chi}).
Datalog Evaluation

• Bottom-up Evaluation
  – Start with the EDB relations
  – Apply rules till convergence
  – Construct the minimal Herbrand model

• Top-down Evaluation (Prolog)
  – Start with a query
  – Construct a proof tree by left to right-evaluation
  – Limited resolution
The “Naïve” Evaluation Algorithm

1. Start by assuming all IDB relations are empty
2. Repeatedly evaluate the rules using the EDB and the previous IDB, to get a new IDB
3. Terminate when no change to IDB
Naïve Evaluation

flight(sfo, den).
flight(sfo, dal).
flight(den, chi).
flight(den, dal).
flight(dal, chi).

reachable(X, Y) :- flight(X, Y).
reachable(X, Y) :- reachable(X, Z), flight(Z, Y).
reachable(sfo, den).
flight(sfo, den).
flight(sfo, dal).
flight(den, chi).
flight(den, dal).
flight(dal, chi).

reachable(sfo, den).
reachable(sfo, dal).

reachable(X, Y) :- flight(X, Y).
reachable(X, Y) :- reachable(X, Z), flight(Z, Y).
Naïve Evaluation

flight(sfo, den).
flight(sfo, dal).
flight(den, chi).
flight(den, dal).
flight(dal, chi).

reachable(X, Y) :- flight(X, Y).
reachable(X, Y) :- reachable(X, Z), flight(Z, Y).
Naïve Evaluation

flight(sfo, den).
flight(sfo, dal).
flight(den, chi).
flight(den, dal).
flight(dal, chi).

reachable(sfo, den).
reachable(sfo, dal).
reachable(den, chi).
reachable(den, dal).
reachable(dal, chi).

reachable(X, Y) :- flight(X, Y).
reachable(X, Y) :- reachable(X, Z), flight(Z, Y).
Naïve Evaluation

flight(sfo, den).
flight(sfo, dal).
flight(den, chi).
flight(den, dal).
flight(dal, chi).

reachable(X, Y) :- flight(X, Y).
reachable(X, Y) :- reachable(X, Z), flight(Z, Y)

reachable(sfo, den).
reachable(sfo, dal).
reachable(den, chi).
reachable(den, dal).
reachable(dal, chi).
reachable(sfo, chi).
Seminaïve Evaluation

- Key idea: to get a new tuple for relation $P$ on one round, the evaluation must use some tuple for some relation of the body that was obtained on the previous round
- Maintain $\Delta P = \text{new tuples added to } P \text{ on previous round}$
- “Differentiate” rule bodies to be union of bodies with one IDB subgoal made “$\Delta$.”
Seminaïve Evaluation

flight(sfo, den).
flight(sfo, dal).
flight(den, chi).
flight(den, dal).
flight(dal, chi).

reachable(sfo, den).
reachable(sfo, dal).
reachable(den, chi).
reachable(den, dal).
reachable(dal, chi).

reachable(X, Y) :- flight(X, Y).
reachable(X, Y) :- reachable(X, Z), flight(Z, Y).
flight(sfo, den).
flight(sfo,dal).
flight(den, chi).
flight(den, dal).
flight(dal, chi).

reachable(X, Y) :- flight(X, Y).
reachable(X, Y) :- reachable(X, Z), flight(Z, Y).

reachable(sfo, den).
reachable(sfo, dal).
reachable(den, chi).
reachable(den,dal).
reachable(dal,chi).
reachable(sfo,chi).
Summary Bottom-Up Evaluation

• Efficient
• Useful
  – Databases
  – Program analysis
  – Semantic web
  – Network programming
• Tools
  – Java Iris
  – pyDatalog
  – Inter4Q (C++)
• But limited
Prolog

- Short for Programmation en logique
- Created by Alain Colmerauer in 1972 during research on Natural Language Processing
- A Turing complete language
  - Beyond Datalog
  - Not guaranteed to terminate
- Employs a goal-directed, backtracking depth-first search to answer queries in top-down evaluation to answer queries
  - Semantics sensitive to the order of rules and conjunctions
- Public domain version: http://www.swi-prolog.org/
Prolog Search Strategy

• Tries to prove a goal g(...) by investigating each rules of g in order

  g(a).
  g(X) :- f(X), h(X, Y).
  g(X):- ...

• Conjuncts are proved left-to-right
Example Flight Planning: “? r(a, X)”

r(a, X)

Solutions

f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- f(S, Z), r(Z, Y).
Example Flight Planning: “? r(a, X)”

r(a, a)

Solutions

r(a, a)

f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- f(S, Z), r(Z, Y).
Example Flight Planning: “? r(a, X)”

```
f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- f(S, Z), r(Z, Y).
```

**Solutions**

```
r(a, a)
```
Example Flight Planning: “? r(a, X)”

\[ f(a, b). \]
\[ f(a, c). \]
\[ r(S, S). \]
\[ r(S, Y) :- f(S, Z), r(Z, Y). \]
Example Flight Planning: “? r(a, X)”

f(a, b).

f(a, c).

r(S, S).

r(S, Y) :- f(S, Z), r(Z, Y).

Solutions

r(a, a)
r(a, b)
Example Flight Planning: “? r(a, X)”

r(a, X)

Solutions
r(a, a)
r(a, b)

r(a, a)
f(a, Z), r(Z, X)

f(a, Z)

r(b, X)

r(b, b)
f(b, Z), r(Z, X)

f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- f(S, Z), r(Z, Y).
Example Flight Planning: “? r(a, X)”

Solutions
- r(a, a)
- r(a, b)

```
f(a, Z), r(Z, X)
f(a, Z)
f(a, b)
r(b, Z), r(Z, X)
r(b, b)
```

```
f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- f(S, Z), r(Z, Y).
```
Example Flight Planning: “? r(a, X)”

```
f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- f(S, Z), r(Z, Y).
```

Solutions

- r(a, a)
- r(a, b)
Example Flight Planning: “? r(a, X)"

```
f(a, b).  
f(a, c).  
r(S, S).  
r(S, Y) :- f(S, Z), r(Z, Y).```

Solutions

r(a, a)  
r(a, b)  
r(a, c)
Example Flight Planning: “? r(a, X)”

Solutions
- r(a, a)
- r(a, b)
- r(a, c)

f(a, b).

f(a, c).

r(S, S).

r(S, Y) :- f(S, Z), r(Z, Y).
Computation and Proof Search

- We can think of proof search as performing computation.
- The inputs and outputs of the computation happen through assignments made to free variables.
Will Prolog’s proof search always work?

• Old example

```prolog
f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- f(S, Z), r(Z, Y).
```

• New example

```prolog
f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- r(S, Z), f(Z, Y).
```
Example Flight Planning: “? r(a, X)"

f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- r(S, Z), f(Z, Y).

Solutions
r(a, a).
Example Flight Planning: “? r(a, X)”

```
f(a, b).
f(a, c).
r(S, S).
r(S, Y) :- r(S, Z), f(Z, Y).
r(a, X) :- r(S, Y).
r(S, Y) :- r(S, Z), f(Z, Y).
r(a, a).
r(a, Y).
r(a, Z).
r(a, X).
```

Infinite loop
Soundness and Completeness

• Prolog's proof search procedure is sound but not complete
  – Sound
    • If we can prove something, then it is true in the logical reading of the program
  – Complete
    • We can prove everything that is logically true
Soundness and Completeness: Garbage Collection

• A garbage collection algorithm can be thought of as proving that blocks of memory are garbage

• Example: treat elements which are not reachable from the roots (reference variables and globals)
  – Sound?
  – Complete?
Terms

• The values of Prolog are called terms
• The values a, b, sfo, den, . . . are a special type of term called an atom
• Variables like X, Y are also a type of term
Compound Terms

- Prolog supports terms beyond Datalog
- Can build compound terms
- Syntax: function_symbol(t₁, t₂, ..., tₙ)
- Examples:
  - f(sfo)
  - f(f(sfo))
  - g(sfo, f(den))
- Lists
  - Special kind of terms
  - [A, B, C]
  - [A, B, C | Tail]
- Terms are not the same as predicates
  - Term: a value: a name of something
  - Predicate: Either true or false
Matching Terms

• Previously we glossed over what it meant for two terms to match because it was obvious:
  – sfo = sfo ✔
  – dal = dal ✔
  – den = dal ✗
  – sfo = chi ✗
Matching Compound Terms

• What happens in the case of compound terms?
  – \( f(X) = f(3) \)
  – \( f(X) = f(f(Y)) \)
  – \( g(X, Y) = f(3) \)
  – \( g(X, Y) = g(Z, Z) \)
  – \( g(f(3), Y) = g(h(X), Z) \)
Matching Compound Terms

- In Prolog we say two terms are unifiable if we can apply a variable substitution such that the two terms become the same.
  - \( f(X) = f(3) \)  \( \checkmark \) \( \{X = 3\} \)
  - \( f(X) = f(f(Y)) \)  \( \checkmark \) \( \{X = f(Y)\} \)
  - \( g(X, Y) = f(3) \)  \( \times \)
  - \( g(X, Y) = g(Z, Z) \)  \( \checkmark \) \( \{X = Z, Y = Z\} \)
  - \( g(f(3), Y) = g(h(X), Z)) \)  \( \times \)
Proofs and Unification

- Lets illustrate how proofs and unification interact using a list membership predicate `mem`
- \( \text{mem}(\text{Ys}, \text{X}) \) holds if \( \text{X} \) is a member of list \( \text{Ys} \)

\begin{verbatim}
% base case
\text{mem}([\text{X}|\_], \text{X}).

% recursive case
\text{mem}([\_|\text{Ys}], \text{X}) :- \text{mem}(\text{Ys}, \text{X}).
\end{verbatim}
List Membership

**mem([1, 2], Z)**

**Solutions**

```prolog
mem([X|_], X).
mem([_|Ys], X) :- mem(Ys, X).
```
List Membership

mem([1, 2], Z)

Solutions
mem([1, 2], 1)

X=1, X = Z

mem([1, 2],1)

mem([X|__], X).

mem([__|Ys], X) :- mem(Ys, X).
List Membership

mem([1, 2], Z).

mem([X|_], X).

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

Solutions

mem([1, 2], 1)

X = 1, X = Z

Ys = 2, Z = X

mem([1, 2], 1)

mem([1, 2], Z)
List Membership

mem([1, 2], Z).
mem([1, 2], X).
mem([1, 2], 1).
mem([1, 2], 2).
mem([2], X).
mem([2], 1).
mem([2], 2).
mem([X|_], X).
mem([_|Ys], X) :- mem(Ys, X).

Solutions

mem([1, 2], 1)
mem([1, 2], 2)
List Membership

mem([1, 2], Z).
mem([X|_], X).
mem([_|Ys], X) :- mem(Ys, X).
mem([1, 2], 1).
mem([1, 2], 2).

Solutions
mem([1, 2], 1)
mem([1, 2], 2)
List Membership

mem([1, 2], Z).

mem([X|_], X).

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

Solutions

mem([1, 2], 1)
mem([1, 2], 2)
Directionality

• We've seen that mem([1, 2], X) returns 1 and 2

• What happens for mem(X, 42)?
  – X = [42 | _Z1]
  – X = [ _Z1, 42 | _Z2]
  – ...

• Sometimes inversions is useful reachable(X, a).
Negation

• Negation can be useful
• Example: to compute all pairs of disconnected nodes in a graph

reachable(S,D) :- link(S,D).
reachable(S,D) :- link(S,Z), reachable(Z,D).
unreachable(S,D) :- node(S), node(D), \+reachable(S,D).

• Prolog makes a closed-world assumption. That is, things that we cannot prove true are false
• \+ mem([1, 2], 3)
• Datalog usually enforces further restrictions
Arithmetic

- Prolog supports arithmetic on ground terms

%base case
length([], 0).

% recursive case
length([H|T], L) :- length(T, L1),
    L is L1 + 1.
N-Queens in Prolog

diagsafe(_, _, []).

diagsafe(Row, ColDist, [QR|QRs]) :- RowHit1 is Row + ColDist, QR =\= RowHit1,
                  RowHit2 is Row - ColDist, QR =\= RowHit2,
                  ColDist1 is ColDist + 1, 
diagsafe(Row, ColDist1, QRs).

safe_position([_]).
safe_position([QR|QRs]) :- diagsafe(QR, 1, QRs), safe_position(QRs).

nqueens(N, Y) :- sequence(N, X), permute(X, Y), safe_position(Y).
Summary Prolog

• Can be powerful when used carefully
  – Inductive definitions
  – Parsers
  – Type checkers
  – Solving games
  – Search problems
• Implements a special case of resolution
• Includes primitive to restrict backtracking and side-effects (cut, fail)
  – Can be hard to understand
Summary Prolog vs. Datalog

• Datalog is more restricted
  – Restricted compound terms
  – Restricted negations
    • Stratified
  – Safe rules
    • Every variable in the rule must occur in a positive (non-negated) relational atom in the rule body

• But easier to understand
• More declarative
• Simpler semantics