Prolog

Reading: Sethi, Chapter 11.

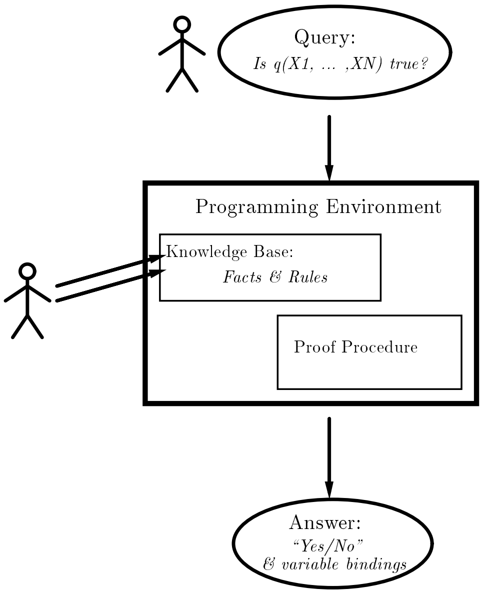
- Overview
- Predicate Calculus
- Substitution and Unification
- Introduction to Prolog
- Prolog Inference Rules
- Programming in Prolog
  - Recursion
  - List Processing
  - Arithmetic
  - Higher-order programming
  - Miscellaneous functions
- Conclusion

Prolog

Programming in **Log**ic

- idea emerged in early 1970's;
  most work done at Univ. of Edinburgh
- based on a subset of first-order logic
  - feed it theorems and pose queries, system does the rest
- main uses:
  - originally, mainly for natural language processing.
  - now finding uses in database systems and even rapid prototyping systems of industrial software.

Logic Programming Framework



Declarative Languages

Also known as logic programming.
Popular languages: Prolog, SQL, Datalog.
Typical of database systems and artificial intelligence.

Declarative specifications: Specify what you want, but not how to compute it.

Example. Find `X` and `Y` such that

```
3 X + 2 Y = 1
X - Y = 4
```

A method (program) for solving these is *how* to get values for `X` and `Y`. But all we gave was a *specification*, or *declaration* of what we want. Hence the name.

Examples

• "Retrieve the telephone number of the person whose name is Tom Smith" (easy)

• "Retrieve the telephone number of the person whose address is 13 Black St" (hard)

• "Retrieve the name of the person whose telephone number is 123-3445" (hard)

Each command specifies *what* we want but not *how* to get the answer. A database system would use a different algorithm for each of these cases.

Can also return multiple answers:
• "Retrieve the names of *all* people who live on Oak St."

Algorithm = Logic + Control

• Users specify "logic" — *what* the algorithm does — using logical rules and facts.

• "Control" —*how* the algorithm is to be implemented — is built in to Prolog.

i.e., search procedures are built in. They apply rules in a particular order to answer user questions.

Example. `P if` Q_1 `and` Q_2 `and ... and` Q_k can be read as

to deduce P:
  deduce Q_1
  deduce Q_2
  ...
  deduce Q_k

Users specify what they want using classical first-order logic (predicate calculus).

Classical First-Order Logic

• simplest form of logical statements is an *atomic formula*. *e.g.*,

`man(tom)` (tom is a man)

`woman(mary)` (mary is a woman)

`married(tom,mary)`
(tom and mary are married)

More complex formulas can be built up using *logical connectives*: $\wedge$, $\vee$, $\sim$, $\forall X$, $\exists X$. *e.g.*,

`smart(tom)` $\vee$ `dumb(tom)`

`smart(tom)` $\vee$ `tall(tom)`

$\sim$ `dumb(tom)`

$\exists$`X married(tom, X)`
(tom is married to something)

$\forall$`X loves(tom, X)`
(tom loves everything)

$\exists$`X [married(tom, X)` $\wedge$ `female(X)` $\wedge$ `human(X)]`
(tom is married to a human female)

Logical Implication

`rich(tom)` $\vee$ $\sim$`smart(tom)`

This implies that if tom is smart, then he must be rich. So, we often write this as

`rich(tom)` $\leftarrow$ `smart(tom)`

In general, $P \leftarrow Q$ and $Q \rightarrow P$ are abbreviations for $P \vee \sim Q$.

For example,

$\forall$`X [(person(X)` $\wedge$ `smart(X))` $\rightarrow$ `rich(X)]`
(every person who is smart is also rich)

$\exists$`X mother(john,X)`
(john has a mother)

$\exists$`X [mother(john,X)` $\wedge$
  $\forall$`Y mother(john,Y)` $\rightarrow$ `Y = X]`
(john has *exactly* one mother)

Horn Rules

Logic programming is based on formulas called Horn rules. These have the form

$\forall x_1...x_k\ [A \leftarrow B_1 \wedge B_2... \wedge B_j]$

where $k, j \geq 0$.

For example,

$\forall$X,Y [A(X) $\leftarrow$ B(X,Y) $\wedge$ C(Y)]
$\forall$X [A(X) $\leftarrow$ B(X)]
$\forall$X [A(X,d) $\leftarrow$ B(X,e)]
A(c,d) $\leftarrow$ B(d,e)
$\forall$X A(X)
$\forall$X A(X,d)
A(c,d)

Note that atomic formulas are also Horn rules, often called facts.

A *set* of Horn rules is called a *Logic Program*

Logical Inference with Horn Rules

Logic Programming is based on a simple idea:
From rules and facts derive more facts

Example 1. Given the facts A, B, C, D, and these rules:

(1) E $\leftarrow$ A $\wedge$ B
(2) F $\leftarrow$ C $\wedge$ D
(3) G $\leftarrow$ E $\wedge$ F

¿From (1), derive E
¿From (2), derive F
¿From (3), derive G

Example 2. Given these facts:

man(plato) ("plato is a man")
man(socrates) ("socrates is a man")

and this rule:

$\forall$X [man(X) $\rightarrow$ mortal(X)]
("all men are mortal")

derive: mortal(plato), mortal(socrates).

Recursive Inference

Example.

Given:

$\forall$X [mortal(X) $\rightarrow$ mortal(son_of(X))]
mortal(plato)

Derive:

mortal(son_of(plato))
(using X = plato)

mortal(son_of(son_of(plato)))
(using X = son_of(plato))

mortal(son_of(son_of(son_of(plato))))
(using X = son_of(son_of(plato)))

...

This kind of inference simulates recursive programs (as we shall see).

Logic Programming

Horn rules correspond to programs, and a form of Horn inference corresponds to execution.

For example, consider the following rule:

$\forall$X,Y p(X) $\leftarrow$ q(X,Y) $\wedge$ r(X,Y) $\wedge$ s(X,Y)

Later, we shall see that this rule can be interpreted as a program, where

p is the program name,
q,r,s are subroutine names,
X is a parameter of the program, and
Y is a local variable.

Non-Horn Formulas

The following formulas are *not* Horn:

```
A → ∼ B
A ∨ B
A ∨ B ← C
∃X [A(X) ← B(X)]
A ← (B ← C)
∀X [flag(X) → [red(X) ∨ white(X)]]
```
("every flag is red or white")

```
∀X ∃Y [wife(X) → married(X,Y)]
```
("every wife is married to someone")

Non-Horn Inference

Inference with non-Horn formulas is more complex than with Horn rules alone.

Example.

```
A ← B
A ← C
B ∨ C
```
(non-Horn)

We can infer `A`, but must do case analysis:

```
either B or C is true.
  if B then A
  if C then A
Therefore, A is true in all cases.
```

Non-Horn formulas do not correspond to programs, and non-Horn inference does not correspond to execution.

Logical Equivalence

Many non-Horn formulas can be put into Horn form using two methods:

(1) logical equivalence
(2) skolemization

Example 1. Logical Equivalance.

∼A ← ∼B	≡	∼A ∨ ∼(∼B)
	≡	∼A ∨ B
	≡	B ∨ ∼A
(Horn)	≡	B ← A

Logical Laws:

∼∼A	≡	A
∼ (A ∨ B)	≡	∼A ∧ ∼B
A ∨ (B ∧ C)	≡	(A ∨ B) ∧ (A ∨ C)
A ← B	≡	A ∨ ∼B

Example 2. Logical Equivalance.

A ← (B ∨ C)	≡	A ∨ ∼(B ∨ C)
	≡	A ∨ (∼B ∧ ∼C)
	≡	(A ∨ ∼B) ∧ (A ∨ ∼C)
(Horn)	≡	(A ← B) ∧ (A ← C)

Example 3. Logical Equivalence.

A ← (B ← C)	≡	A ∨ ∼(B ← C)
	≡	A ∨ ∼(B ∨ ∼C)
	≡	A ∨ (∼B ∧ ∼∼C)
	≡	A ∨ (∼B ∧ C)
	≡	(A ∨ ∼B) ∧ (A ∨ C)
(non Horn)	≡	(A ← B) ∧ (A ∨ C)

In general, rules of the following form *cannot* be converted into Horn form:

$$\forall x[(A_1 \vee ... \vee A_n) \leftarrow (B_1 \wedge ... \wedge B_m)]$$

For example,

```
(A ∨ B) ← (C ∧ D)
(A ∨ B) ← C
(A ∨ B)
∀X [A(X) ∨ B(x)] ← [C(X) ∧ D(X)]
```

$i.e.$, if it is possible to infer a *non-trivial* disjunction from a set of formulas, then the set is inherently non-Horn.

(A rule like $p \vee q \leftarrow q$ infers a *trivial* disjunction, since the rule is a logical tautology. Such rules can simply be ignored.)

Skolemization

Non-Horn formulas like $\exists x\ A(x)$ can be converted to Horn form.

Example 1.

Replace (1)	∃X mother(john,X)	(non-Horn)
with (2)	mother(john,m)	(Horn)

Here, m is a new constant symbol, called a *skolem constant*, that stands for the (unknown) mother of john.

Note: (1) $\not\equiv$ (2), but they say (almost) the same thing. In particular, (1) can sometimes be replaced by (2) during inference, as we shall see.

Skolemization (Cont'd)

Example 2. A non-Horn formula:

(3) ∀X [person(X) → ∃Y mother(X,Y)]
"every person has a mother")

Let m(x) stand for the (unknown) mother of X. Then, we can replace (3) by a Horn rule:

(4) ∀X [person(X) → mother(X,m(X))]

m(X) is called a *skolem function*.
It is an artificial name we have created.
e.g., m(mary) denotes the mother of mary.
m(tom) denotes the mother of tom.
m(jfk) denotes the mother of jfk.

So, we only need ∃X because we don't have a *name* for X. By creating artifical names (skolem symbols), we can eliminate many ∃'s, and convert many formulas to Horn rules, which Prolog can then use.

Skolemization is a technical device for doing inference.

Inference with Skolemization

(1) ∀X [man(X) → person(X)]
("every man is a person")

(2) ∀X ∃Y [person(X) → mother(X,Y)]
("every person has a mother"—non Horn)

(3) ∀X,Y [mother(X,Y) → loves(Y,X)]
("every mother loves her children")

(4) man(plato) ("plato is a man")

Question. ∃Y loves(Y,plato)
("does someone love plato?")

Step 1. Skolemize (2) to get a Horn rule:
(2') ∀X [person(X) → mother(X, m(X))]

Step 2. Use Horn inference:

person(plato)	from (1)
mother(plato,m(plato))	from (2')
loves(m(plato),plato)	from (3)

Thus. ∃Y loves(Y,plato)
i.e., Y = m(plato). So, answer is YES.

Skolem Dependencies

(1) ∃X ∀Y p(X,Y)

skolemizes to ∀Y p(a,Y),
where a is a skolem constant.

(2) ∀Y ∃X p(X,Y)

skolemizes to ∀Y p(b(Y),Y),
where b is a skolem function.

i.e., in (2), X depends on Y.
But in (1), X is independent of Y.

(3) ∀X ∀Y ∃Z q(X,Y,Z)

skolemizes to ∀X ∀Y q(X,Y,c(X,Y)),
where c is a skolem function of both X and Y.

i.e., in (3), Z depends on both X and Y.

Skolem Dependencies — Concrete Examples

$\exists$X $\forall$Y `loves(X,Y)` ("someone loves everybody")

$\Rightarrow$ $\forall$Y `loves(p,Y)` ("p loves everybody")

$\forall$X $\exists$Y `mother(X,Y)` ("everyone has a mother")

$\Rightarrow$ $\forall$X `mother(X,m(X))`
("m(X) is the mother of X")

$\forall$X $\forall$Y $\exists$Z `owns(X,Y)` $\rightarrow$ `document(Z,X,Y)`

("if X owns Y, then there is a document, Z, saying that X owns Y")

$\Rightarrow$ $\forall$X $\forall$Y `owns(X,Y)` $\rightarrow$ `document(d(X,Y),X,Y)`

("d(X,Y) is a document saying that X owns Y")

Prolog

- Syntax: — Horn clauses of *terms*
- Proof process: <u>*SLD Resolution*</u>
  - *unification* + *substitution*:
    pattern matching between terms + binding unresolved variables as needed.
  - *automatic backtracking*:
    if one attempt fails, try again until all search paths exhausted.

Prolog Notation

For convenience, we often don't write universal quantifiers, so the rule

$\forall$X `[p(X)` $\leftarrow$ `(q(X)` $\wedge$ `r(X))]`

is written as

`p(X)` $\leftarrow$ `q(X), r(X).`

We will also use the Prolog conventions:

- *variables* begin in upper case.
  e.g., `A, B, X, Y, Big, Small, ACE`
- *constants* begin in lower case.
  e.g., `a, b, x, y, abbot, costello`

Database Queries

Consider the following database of facts:

`loves(bob,sue)`
`loves(sue,tony)`
`loves(sue,harry)`
`loves(X,santa)` *i.e.*, $\forall$X `loves(X,santa)`
i.e., "everyone loves santa"
`loves(jesus,X)` *i.e.*, $\forall$X `loves(jesus,X)`
i.e., "jesus loves everyone"

We would like to pose questions (queries) to this database.

e.g., "Who does sue love?"
i.e., Find Y such that loves(sue,Y) is true.
answers: Y = tony
Y = harry
Y = santa

"Who loves sue?"

i.e., Find Y such that loves(Y,sue) is true.

answers: Y = bob
Y = jesus

A Conjunctive Query.

"Find someone who loves tony *and* who is loved by bob."

i.e., Find Y such that loves(Y,tony) and loves(bob,Y) are both true.

answer: Y = sue

A Disjunctive Query.

"Find someone who loves tony *or* who is loved by bob."

i.e., Find Y such that loves(Y,tony) or loves(bob,Y) is true.

answers: Y = sue
Y = jesus
Y = sue
Y = santa

To answer queries, Prolog uses Unification.

food for slide eater

food for slide eater

food for slide eater