CSci 8980, Fall 2012
Specifying and Reasoning About Computational Systems
An Introduction to Logic Programming

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2012

The Idea Underlying Logic Programming

- The key idea is to use a specification of a program directly as a means for computing
  - define relations in a specification language
  - try to deduce relevant relations from the specification
  - if successful, provide a corresponding answer
- The idea is workable only if we have a specification language with the following properties
  - it is sufficiently expressive
  - deduction in it can be connected up with computation in a useful way
    - Can computational behaviour also be specified?
    - Can the idea of a "result" be interpreted sensibly?
  - it is efficiently implementable

First-order Horn clause logic and Prolog represent one realization of this idea

Hallmarks of (First-Order) Logic Programming

- Programming/specification follows a *relational* style

  For example, *append* is specified as a relation between three lists

  Moreover, we compute by asking questions about *append* as a relation
- Objects are represented by using a general category of expressions called *first-order terms*

  Similar to values without function components in functional programming, generalized to contain variables
- *Unification* provides a central means for decomposing object representations
- The programming interpretation provides a natural realization of search

Model of Computation in Logic Programming

- Define a database of relations between objects via
  - *facts*

    For example like "John likes Mary"
  - *rules*

    E.g. "John likes a person if that person likes Mary"
- Ask questions against such a database

  E.g. "Does John like Mary?"

  Such expressions are called *queries* or *goals*

  Goal solving is the counterpart in logic programming to expression evaluation in functional programming

The "database" example and view is only one possibility; complex computations can also be described and executed

Representing Relations

Relations are represented in *intension* via predicates

$\langle$*Name-of-rel*$\rangle(\langle$*Object*$_1\rangle, \ldots, \langle$*Object*$_n\rangle)$

Here

- $\langle$*Name-of-rel*$\rangle$ represents the name of the relation
- $\langle$*Object*$_i\rangle$ is a description of an object

An example of a relation

likes(john,mary)

Think of this as saying $\langle$*john,mary*$\rangle \in$ *likes*

Syntax Conventions from Prolog

- Constants will be represented by tokens beginning with a lower case letter or a special symbol
- For now, names of relations will be expected to be constants
- The simplest object description is also a constant

Defining Relations Through Facts

Facts are represented by a relation followed by a period such as the following

```
likes(john,mary).
likes(mary,john).
likes(john,bill).
likes(sue,mary).
likes(mary,bill).
likes(sue,bill).
```

Facts can also contain variables (tokens starting with uppercase letters) representing universal quantification, e.g.

```
likes(tom,X).
```

Once you have a database set up you can query it:

```
?- likes(john,mary).
true.
?-
```


Some Variants on the Simple Query Form

- Queries can also be conjunctive:

  ```
  ?- likes(john,mary), likes(mary,john).
  ```

  A conjunctive goal is true if both subgoals are true
- Queries can have *variables*, representing existential quantification, in them

  ```
  ?- likes(john,X).
  ```

  The interpretation: *"find me something that makes the query true"*

  Thus, queries with variables can have multiple solutions

  The way Prolog solves such queries
  - Sequence through database looking for a match
  - If success is encountered, leave a place marker and return the successful match
  - If success is rejected, resume from place marker

Variations on the Simple Query Form (Contd)

We can mix variables and conjoined forms of queries too

```
?- likes(john,X), likes(mary, X).
```

Prolog solves conjunctive goals G_1, G_2 with variables as follows:

- Try to solve G_1
- If this attempt is successful
  - instantiate the variables in G_1 and G_2 as per the match
  - leave a place marker to return to if G_2 fails
  - Try to solve G_2
  - If the attempt to solve G_2 is successful
    - leave a place marker to return to if answer is rejected
    - Provide the instantiations as an answer
  - If the attempt to solve G_2 fails, backtrack to G_1
- If the attempt to solve G_1 fails then fail overall

Defining Relations Through Rules

The general structure of rules is the following

⟨*head-pred*⟩ `:-` ⟨*body-goal*⟩.

The symbol `:-` should be read as "if"

Rules allow for the statement of conditioned facts

E.g., "John likes a person if that person likes Mary" becomes

```
likes(john,X) :- likes(X,mary).
```

Some further things to note about rules

- Variables in rules should be thought of as being universally quantified with the entire rule as the scope
- Rules can be thought of as (partial) procedure declarations

  head of rule ↔ procedure header
  body of rule ↔ procedure body

  Partial because a predicate can have more than one rule

Representations of Data Objects

Determined by what can appear as the arguments of predicates

In Prolog, these are *first-order terms* defined as follows

- *Variables*
  - tokens beginning with upper case letters
  - _, for the anonymous variable
- *Constants*
  - tokens beginning with lower case letters
  - Special sign tokens like `=`, `+`, `:-`, `<`, `?-`, etc
  - String, integer,real, etc, constants represented using some chosen syntactic convention
- *Compound terms*, constructed from other terms using function symbols

  ⟨*constant*⟩(*term*$_1, \ldots,$ term$_n$)

  Some terminology: *principal functor*, *arity*, *arguments*
  Some of this terminology carries over to predicates

The Expressiveness of First-Order Terms

Without considering variables, first-order terms can already represent any hierarchical structure

For example, a person record can be given as follows

```
person(name('John', 'Smith'),
       ssno('351609659'),
       dob(month(feb),
           day(2),
           year(1960)))
```

Can be visualized as a tree, making explicit the hierarchical structure

First-order terms subsume non-function expressions in functional languages like ML, Haskell and OCaml

Although we have not assumed this yet, typing can easily be introduced within first-order terms

Recap: First-Order Terms as Data Structures

- The data structures of a logic programming language are what can appear as arguments of atomic predicates
- In the setting of Prolog, these are *first-order terms*

  Formed from variables and (designated) constants using (designated) function symbols

  We have assumed an untyped signature for the moment, but the signature can be typed too
- Any hierarchical structure, i.e. data representable using structures in C, can be encoded by these terms

The Expressiveness of First-Order Terms

With variables thrown into the mix, first-order terms become *more general* than hierarchical structures

- They can be used to represent *classes* of such structures

  For example, consider
  `person(name(_,'Smith'),_,_)`

  Like *patterns* in functional languages, except that these can be used on the *input* side as well

- Actually, variables can be repeated in terms, thereby providing a means for representing *constraints*

  For example, consider `tree(_,X,X)`

  Graphically, this means first-order terms generalize trees (hierarchical structures) to digraphs

  This kind of possibility is typically explicitly ruled out in the functional programming setting

Recursive Data and First-Order Terms

The recursive structure of terms make them very natural for representing recursive objects

For example, consider lists

- empty lists can be represented by the constant *nil*
- non-empty lists can use the function symbol `::`

E.g., `::(1,::(2,::(3,nil)))` represents a list

Similarly, consider trees

- empty trees can be represented by the constant *empty*
- non-empty trees can use the function symbol *node*

E.g., `node(3,node(2,empty,empty),empty)`

Lists are ubiquitous and so often have a special notation

`nil`	$\leftrightarrow$	`[]`
`::(h,t)`	$\leftrightarrow$	`[h | t]`
`::(e1,::(e2,...,::(en,nil)))`	$\leftrightarrow$	`[e1,e2,...,en]`

Unification of First-Order Terms

First-order unification is the main operation for analyzing terms

The issues that define this operation

- Do two terms t_1 and t_2 have a common instance?
- What is the substitution for variables that generates this common instance?

The substitution is what is known as a *unifier*

We are typically interested in *most general unifiers*, i.e. unifiers from which others can be generated by a further substitution

For example, consider $f(X)$ and $f(g(Y))$

Theoretically, unification can be viewed as equation solving over a *set* of equations between terms

$\{t_1 = s_1, \ldots, t_n = s_n\}$

Some Examples of Unification Problems

- $\{f(X, g(X)) = f(g(Y), g(g(h(Z))))\}$
  An mgu here is $\{Y = h(Z), X = g(h(Z))\}$

  Note that a most general unifier (*mgu*) can be seen as a unification problem in a special form

- $\{f(X, g(X)) = f(g(a), g(h(a)))\}$
  Here there is a *clash* failure because X cannot be $g(a)$ and $h(a)$ simultaneously

- $\{f(X, Y) = f(g(Y), g(X))\}$

  Here there is an *occurs-check* failure: Y cannot equal $g(g(Y))$ for finite terms

  Most Prolog implementations ignore the occurs-check, justifying it by assuming terms can be infinite

  However, finiteness of terms is *essential* for logical applications and we will therefore insist on it

An Algorithm for Finding (Most General) Unifiers

The idea is to incrementally transform a given unification problem into a solved form

A unification problem is in solved form if

- the lefthand sides of each equation is a variable
- a variable that appears as the lefthand side of an equation appears only there

Exercise: Show that a unification problem in solved form is its own most general unifier

A (non-deterministic) unification algorithm can be described by presenting transformations that

- simplify problems in the sense of getting them closer to a solved form
- preserve the set of unifiers while simplifying the problem

For first-order unification, three transformations called *variable elimination*, *reorientation* and *term reduction* do the trick

The Reorientation Transformation

This is a bookkeeping transformation whose purpose is to get variables to the lefthand sides of equations

Specifically, the transformation is the following

> *Replace an equation of the form $t = x$, where x is a variable and t is not by $x = t$*

The Term Reduction Transformation

This transformation simplifies the top-level "rigid" structure of terms in equations

The transformation applies to an equation of the form

$$f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$$

where f and g are constants or function symbols

- If f and g are *different* symbols, replace the unification problem with the unsolvable one denoted by $\perp$
- Otherwise, if $f = g$ and hence $m = n$, replace the equation under consideration by the equations

$$t_1 = s_1, \ldots t_n = s_n$$

Exercise: Show that the transformation preserves the set of unifiers

The Variable Elimination Transformation

This transformation tries to remove occurrences of a variable that appears as the lefthand side of an equation

It applies when there is an equation of the form $x = t$ where x is a variable that appears elsewhere in the equations

- If t is x, then simply remove the equation
- Otherwise, if x appears in t, then replace the unification problem with $\perp$
- Otherwise, transform the unification problem by replacing x in all *other* equations by t and move $x = t$ to a "solved part"

Exercise: Show the following

- the variable elimination transformation preserves unifiers
- there is a well-founded measure associated with unification problems that is diminished by each transformation

First-Order Unification (Continued)

To get a deterministic unification algorithm, we must impose an order on the application of transformations

The order in which we choose the equations to apply the transformations to makes a *big* difference in complexity

A naive order can result in exponential behaviour, a clever choice can yield a (near) linear algorithm

Conceptually, the order choice determines how much sharing we use in realizing unification, e.g., consider

$$\{X = f(W, W), W = g(a, a)\}$$

Exercises

- Try to implement the unification algorithm in Prolog
- Read the Martelli and Montanari paper to understand especially the complexity details

Uses of Unification in Programming

Unification can be used in different modes in computations

- In the mode of an accessor function

  For example, if `Ls` is an input list with no variables in it

  Then unification can be used to probe its structure

  - Is it a non-empty list? `Ls = [_|_]`
  - Are its first two elements identical? `Ls = [X,X|_]`

- In a destructuring mode

  Let `Ls` be a list with no variables and `H` and `T` be unbound

  Then consider the unification question `Ls = [H|T]`

- In the mode of a constructor function

  Here let `Ls` be unbound and `H` and `T` be instantiated

  Then consider the question `Ls = [H|T]`

Interpreted Functions and Unification

The key issue here can be posed as the following question

Do `2 + 3` *and* `5` *unify?*

The surprising answer is “No”

The essential point: functions are uninterpreted and unification is based on structure

Thus, `2 + 3` matches with

`X + Y`, `X + 3`, and `2 + Y`

but not with `5`

Of course, it is convenient to be able to interpret arithmetic functions sometimes

Prolog has a special predicate called `is` to do this

The goal

```
?- X is 2 + 3, X = 5.
```

succeeds with `X` being bound to `5`

Defining Relations Over Recursive Data

Facts and rules naturally encode definitions over recursive data

For example, the *append* relation between three lists can be defined by recursion on the structure of the first list

- The base case, when the first list is empty, is treated easily by a fact

  ```
  append([],L,L).
  ```

- The recursive case, when the first list is of the form `[X|L]`, has a natural rendition as a rule

  ```
  append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
  ```

This definition is in fact just a rendition of an inference rule presentation of *append* base on the syntax of lists

Given this definition, we can try to derive particular goals

```
?- append([1,2],[3],L).
?- append(L1,L2,[1,2]).
```

More Examples of Relations Over Lists

Consider defining membership in a list

Thus, *member(X,L)* should succeed only if *X* appears in list *L*

- Could hold because *X* is the first element

```
member(X,[X|_]).
```

- Could hold because *X* is a member of the tail

```
member(X,[_|L]) :- member(X,L).
```

Notice that we don't have to say anything to explicitly cover the case when *X* is *not* a member of *L*

The relation can be used in many modes:

```
?- member(1,[2,1,3]).
?- member(X,[2,1,3]).
?- member(1,L).
```


More Examples of Relations Over Lists (Contd)

Consider defining selection from a list

Thus, *select(L1,X,L2)* should succeed if *X* is an element of *L1* and *L2* is the rest

- The first item may be the one we select

```
select([X|L1],X,L1).
```

- The element we select is something other than the first element

```
select([X|L1],Y,[X|L2]) :- select(L1,Y,L2)
```

Notice again, that we do not have to worry about the case where *X* is *not* an element of *L1*

Consider what happens with the query

```
?- select([1,2,3],X,L).
```

Exercise: Use *select* to define a permutation relation between lists

Nondeterministic Programming

Logic programming naturally fits into a style of programming that consists of the following:

- The programmer describes or, more precisely, specifies various operations that are possible next
- The "machine" or "system" is responsible for picking the correct one out of these next operations

This style is often referred to as "non-deterministic" programming

Another name for it: the generate-and-test style

Prolog supports this kind of non-determinism by

- picking the first alternative at any relevant point
- allowing for backtracking if that alternative fails

A Simple Example of Generate-and-Test

Suppose we want to define a relation *inbothlists(X,L1,L2)* that succeeds exactly when *X* is in both *L1* and *L2*

We can structure this as follows:

- Pick an *X* that is in *L1* *(Generate Step)*
- Check if this *X* is in *L2* *(Test Step)*

Thus, the relation can be defined as follows

```
inbothlists(X,L1,L2) :-
     member(X,L1), member(X,L2).
```

As an example, consider how the query

```
?- inbothlists(X,[1,2,4,2],[5,2,7,4]).
```

will be solved

A More Complicated Example

Consider the magic square problem

> *To arrange the numbers from 1 through 9 in a* 3×3 *matrix so that the rows, columns, diagonals and cross diagonals all add up to the same number*

For example, the following is a solution

2	7	6
9	5	1
4	3	8

Using the generate-and-test paradigm, we solve it thus

- Generate an arrangement of the numbers
- Test if the arrangement is a solution

Exercise: Devise a representation for squares of numbers and "implement" this specification approach

Nondeterministic Programming and Efficiency

- The simple approach described significantly decouples generation and testing

  In particular, we generate an entire square before we check it for the property

- We could also integrate the checking of the sum into the generation of the square itself
  - Calculate a sum when the first row is known
  - Rule out further rows if they do not match this sum

- An instance of a general constraint propagation technique that can significantly impact on search behaviour

Exercise: Experiment with this idea in your realization of the magic square specification

Logic versus Programming in Prolog

Logic programming emphasizes the *specification* of relations in the first instance

However, eventually we have also to be concerned about *how* the specifications are going to be used in computation

In particular, the following can make a difference to the efficiency of search in responding to a query:

- the ordering of clauses in a program
- the ordering of goals in a conjunction

In fact, these aspects can sometimes make the difference between a finite and infinite search

Moreover, we may sometimes also want to control the course of search

The Effect of Clause Order on Search

The order in which clauses are presented can impact on the order in which solutions are produced

For example, suppose we have the following facts

```
parent(john,susan).
parent(susan,bill).
parent(bill,jason).
```

Now suppose we have the clauses

```
(1) ancestor(X,Y) :- parent(X,Y).
(2) ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Consider the effect of the order of these clauses on the goal

```
?- ancestor(john,X).
```

Depending on whether (1) appears before (2) or after, we get the closest or farthest ancestor first

Clause Order and Search (Contd)

When there are infinitely many solutions, how you search for one can make a difference between whether you find one or not

For example, suppose we have the clauses

```
(1) is_list([]).
(2) is_list([X|L]) :- is_list(L).
```

We can use these clauses to check if something is a list

```
?- is_list([1,2,3]).
```

We can also use it to generate lists

```
?- is_list(L).
```

The expected lists have one element, two elements, etc

Whether or not Prolog will give us one depends on which list in this sequence we try to return first

The Impact of Goal Order on Search

Different goal orders constrain solution paths differently

Without enough constraints, a (sub)goal may have too many potential solution paths or a neverending one

For example, suppose we have the clauses

```
parent(john,susan).
parent(susan,bill).
parent(bill,jason).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- ancestor(Y,Z), parent(X,Y).
```

Consider then the goal

```
?- ancestor(jason,X).
```

This goal should fail conceptually and does with the earlier ordering of goals in the second `ancestor` clause

However, the present clauses yields an endless search

Redundancy in Search and Pruning

Searching all solution paths is sometimes unnecessary

For example, given the clauses

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).
```

maybe one solution is enough for the goal

```
?- member(a,[a,b,a,c,a]).
```

Specifically, consider its use in the definition

```
inbothlists(X,L1,L2) :-
    member(X,L1), member(X,L2).
```

Multiple solutions are not needed if we use this predicate with all arguments instantiated such as in

```
?- inbothlists(a,[a,b,a,c,a],[b,c,d]).
```

Cut is a Prolog primitive that helps achieve such an effect

Cut: A Primitive for Pruning Search in Prolog

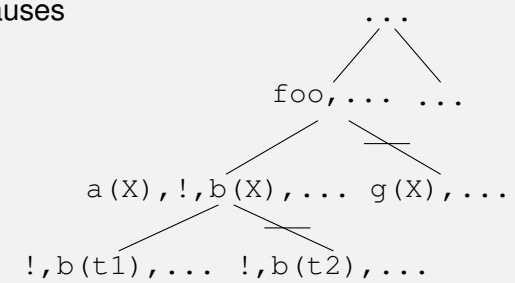
Cut is a "meta-logical" predicate in Prolog that is written `!`

Its interpretation as a goal is as follows:

- it succeeds the first time around
- it cannot be resatisfied
- retrying it causes the parent goal to fail

For example consider the clauses

```
foo :- a(X),!,b(X).
foo :- g(X).
```



If `!` is crossed, trying to backtrack over it causes `foo` to fail

An Example of the Use of Cut

Suppose `member` is defined as follows

```
member(X,[X|L]) :- !.
member(X,[Y|L]) :- member(X,L).
```

Now consider its use in the definition

```
inbothlists(X,L1,L2) :-
    member(X,L1), member(X,L2).
```

In particular, consider again the search behaviour for the goal

```
?- inbothlists(a,[a,b,a,c,a],[b,c,d]).
```

A point to note, however: The cut changes the "specification" of `member` from what was originally intended

Characterizing Uses of Cut

Cuts are typically useful in two modes

- where they are not intended to modify the relations defined but only to change search behaviour
- where they are also intended to play a role in the definition of the relation

The two forms of cut are referred to as *green* and *red*, respectively

Red cuts should be used with care, especially in a setting were we are interested in specifications

Green cuts should also be used with care, lest they turn out to be red cuts!

Green Cuts in Prolog

These are often used in definitions that have the form

```
foo(...) :- <test1>, ...
foo(...) :- <test2>, ...
```

where the two tests are complementary, i.e. they function like a guard for selecting a particular rule

Here we can rewrite the definition as

```
foo(...) :- <test1>, !, ...
foo(...) :- ...
```

For example, this idea can be used to define an if-then-else in Prolog:

```
if(Cond,Then,Else) :- Cond, !, Then.
if(Cond,Then,Else) :- Else.
```

There is a small "cheat" here: we are assuming Prolog has the capability of calling a term as a goal.

But is a Green Cut Really One?

Consider the following definition

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

Is the relation that is defined actually equivalent to

```
min(X,Y,X) :- X =< Y, !.
min(X,Y,Y).
```

Unfortunately, the two definitions do not have identical solution sets

For example, `min(1,2,2)` succeeds with the second definition but not with the first

In other words, the guard in the second clause is essential to keep here

The Not Predicate in Prolog

Prolog provides a "meta-logical" predicate called `not`

`not` takes another predicate and succeeds only when that one fails; this is called *negation-by-failure*

We cannot define `not` logically in Prolog; clauses can only provide *positive* information

However, using a red cut we can define `not`:

```
not(P) :- call(P), !, fail.
not(P).
```

Here `fail` is any predicate that does not have a clause

The `not` predicate is an example of what is called *non-monotonic* reasoning in AI

The Not Predicate in Prolog (Contd)

It is dangerous to use `not` with goals that have variables in them

For example, suppose we have the following clauses

```
student(bill).
married(joe).
unmarried_stud(X) :- not(married(X)),student(X).
```

Now consider the goal

```
?- unmarried_stud(X).
```

Should this succeed? Does it?

The problem arises from the fact that `not(married(X))` is called as a goal before instantiating `X`

Prolog gets confused between existential and universally quantified variables in this setting