Computational Logic

Logic Programming

Georg Moser

Institute of Computer Science @ UIBK

Summer 2015

Summary of Last Lecture

Example (Implementing same_vars)

```
same_var(foo,Y) ← var(Y), !, fail.
same_var(X,Y) ← var(X), var(Y).
```

Example (Bad Cut)

```
minimum(X,Y,X) ← X ⩽ Y, !.        ← minimum(2,5,5)
minimum(X,Y,Y).                   true
```

Types of Red Cuts

1. cuts that are built-in (e.g. in the implementation of negation)
2. green cuts that become red, when conditions are fulfilled
3. supposedly green cut that changes the behaviour of the program

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming, constraint logic programming

Program Access and Manipulation

clause database operations

- *assert/1*

  ```
  ← assert(C).
  true
  ```

- side effect: add rule C to program
- *retract/1*

  ```
  ← retract(C).
  true
  ```

- side effect: remove first rule from program that unifies with C

Example (Fibonacci Numbers Revisited)

```
:- dynamic(fibonacci/2).

fibonacci(0,0).
fibonacci(1,1).
fibonacci(N,X) :-
    N > 1,
    N1 is N-1, fibonacci(N1,Y),
    N2 is N-2, fibonacci(N2,Z),
    X is Y+Z,
    asserta(fibonacci(N,X)),
    !.
```


Example

```
edit :- edit(file([],[])).
edit(File) :-
  read(Command),
  edit(File,Command).
edit(File,exit) :- !.
edit(File,Command) :-
  apply(Command,File,File1),
  !,
  edit(File1).
edit(File,Command) :-
  write(Command),
  write(' is not applicable'),
  !,
  edit(File).
```

```
apply(up,file([X|Xs],Ys),
    file(Xs,[X|Ys])).
apply(down,file(Xs,[Y|Ys]),
    file([Y|Xs],Ys)).
apply(insert(Line), file(Xs,Ys),
    file(Xs,[Line|Ys])).
apply(delete,file(Xs,[Y|Ys]),
    file(Xs,Ys)).
apply(print,file([X|Xs],Ys),
    file([X|Xs],Ys)) :-
  write(X), nl.
apply(print(*),file(Xs,Ys),
    file(Xs,Ys)) :-
  reverse(Xs,Xs1),
  write_file(Xs1),
  write_file(Ys).
```


Operator and Precedences

Query Operator

```
:- current_op(P,A,*).
P ↦ 400,        precedence
A ↦ yfx         infix, left-associative

:- 1*2*3 = (1*2)*3.        :- 1*2*3 = 1*(2*3).
true                       false
```

Define Operator

```
:- op(350, xfy, new).
:- X = *(new(1,*(2,3)),*(4,new(new(4,5),*(6,new(7,8))))).
X ↦ 1 new (2*3) * (4* (4 new 5) new (6*7 new 8))

:- op(450, yfx, new).
:- X = *(new(1,*(2,3)),*(4,new(new(4,5),*(6,new(7,8))))).
X ↦ (1 new 2*3)* (4* (4 new 5 new 6* (7 new 8)))
```


Definition

- if *op*(*Precdence*, *Associativity*, *Name*) is used in program, then it has to be added with `:-`

  `:- op(350,xfy,new)`

- if in a program `:- query` occurs, then `query` is directly executed when the program is loaded
- precedence: positive number, smaller numbers bind stronger
- five modes of associativity
  - xfy: right-associative, `X o Y o Z = X o (Y o Z)`
  - yfx: left-associative, `X o Y o Z = (X o Y) o Z`
  - xfx: non-associative, `X o Y o Z` will not be parsed
  - fy: prefix-operator, `o X`
  - yf: postfix-operator, `X o`

Efficiency of Prolog Programs

Time and Space Complexity

Definition

the time complexity of a (Prolog) program expresses the runtime of a program as a function of the size of its input

Definition

the space complexity of a (Prolog) program expresses the memory requirement of a program as a function of the size of its input

Observations on Space

- space usage depends on the depth of recursion
- space usage depends also on the number of data structures created
- we have already seen that the former may be a major problem: stack overflow

Example

```
sublist(Xs,AXBs) :- suffix(XBs,AXBs), prefix(Xs,XBs).
sublist(Xs,AXBs) :- prefix(AXs,AXBs), suffix(Xs,AXs).
```

Question

What is better?

Answer

the first alternative:

- consider

```
sublist([1,2,3,4],[1,2,3,4,1,2,3,4,1,2,3,4,1,2,3,4])
```

- the 1st clause iterates over the 2nd list to find a suitable suffix
- then iterates over the first list
- no intermediate data structures are created
- in the 2nd clause an auxilliary list is created

Definition

we say: the first clause doesn't cons

Observations on Time

- if full unification (unification of two arbitrary terms in goals) is not employed, reduction of a goal using a clause needs constant time
- that is, it depends only on the program
- hence, if full unification is not employed the number of reductions (= nodes in SLD tree) asymptotically bounds the runtime
- equivalently the number of unifications (performed and attempted) asymptotically bounds the runtime
- on the other hand, if unification needs to be taken into account time complexity analysis is more involved
- in general size of search space and size of input terms needs to be taken into account

Howto Improve Performance

Suggestion ①

use better algorithms ☺

Example

```
reverse([X|Xs],Zs) :-
    reverse(Xs,Ys),
    append(Ys,[X],Zs).
reverse([],[]).
```

Example

```
reverse(Xs,Ys) :- reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) :-
    reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

Suggestion ②

tuning, via:

1. good goal order
2. elimination of (unwanted) nondeterminism by using explicit conditions and cuts
3. exploit clause indexing (order arguments suitably)
   indexing performs static analysis to detect clauses which are applicable for reduction

Example

```
append([X|Xs],Ys,[X|Zs]) :-
    append(Xs,Ys,Zs).
append([],Ys,Ys).
```

By default, SWI-Prolog, as most other implementations, indexes predicates on their first argument.

Recall

- tail recursive programs are called iterative
- reasoning: tail recursion is implemented as iteration which doesn't require a stack

Definition (tail recursion optimisation)

- consider a generic clause for A

$$A' \leftarrow B_1, \dots, B_n$$

  such that A and A' unify with σ
- suppose the goal $B_1\sigma, \dots, B_{n-1}\sigma$ is deterministic
- then goal $B_n\sigma$ can re-use space for A

Definition

clause indexing is used to detect which clauses are applicable for reduction: 2nd clause in append need not be considered

Howto Implement Functions

Functions vs Relations

- often, we want to compute functions:
  1. addition: $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
  2. sorting: $list \to list$
- in logic programming we just specify relations and every function can be seen as a relation

$$f_{rel}(i_1, \dots, i_n, o_1, \dots, o_m) \text{ iff } f(i_1, \dots, i_n) = (o_1, \dots, o_m)$$

- that is, we implement functions $f(i_1, \dots, i_n) = (o_1, \dots, o_m)$ by relations $f_{rel}/(n+m)$
- result is obtained by query $f_{rel}(i_1, \dots, i_n, X_1, \dots, X_m)$
  1. addition: $plus(n, m, Z)$ $\quad Z = n + m$
  2. sorting: $sort(list, Xs)$ $\quad Xs =$ sorted version of $list$

Function Applications

- function applications harder to write down
  - program $f(x) = x^2 + 7 \cdot (x^2 - 5)$
  - defining fact

    `f(X,plus(times(X,X), times(7,minus(times(X,X),5))))`

    does not work
- solution: store result of each sub-expression in fresh variable

```
f(X,Y) :- times(X,X,Z), minus(Z,5,V), times(7,V,U),
plus(Z,U,Y).
```

$$\underbrace{\underbrace{x^2}_{z} + \underbrace{7 \cdot (\underbrace{\underbrace{x^2}_{z} - 5}_{v})}_{u}}_{f(x)=y}$$

Simulating Functional Programs

- using technique of previous slide, it is easy to transform first-order functional programs into logic programs
- remaining difficulty: translating `if-then-else`
  idea: first evaluate condition, and then generate one rule for each branch

Example (Ackermann function in Haskell)

```
ack 0 m = m + 1
ack (n+1) m = if m == 0 then ack n 1 else
          ack n (ack (n+1) (m-1))
```

Example (Ackermann function as logic program)

```
ack(0,M,s(M)).
ack(s(N),M,R) :- =(M,0,B), cond(B,N,M,R).
cond(true,N,M,R) :- ack(N,s(0),R).
cond(false,N,M,R) :- -(M,s(0),U),ack(s(N),U,V),ack(N,V,R).
```


Evaluating Arithmetic Expressions

- motivation: use arithmetic expressions as in functional programs
- solution: write evaluator *eval* which computes value of arithmetic expressions
- afterwards it is very simple to encode functions, e.g.

$$f(x) = s(x^2) - x^2$$

  can be programmed as

  ```
  f(X,Y) :- eval(s(X*X) - X*X, Y).
  ```
- evaluator is simple logic program

  ```
  eval(0,0).
  eval(s(E),s(N)) :- eval(E,N).
  eval(E+F,K) :- eval(E,N), eval(F,M), plus(N,M,K).
  eval(E-F,K) :- eval(E,N), eval(F,M), plus(M,K,N).
  eval(E*F,K) :- eval(E,N), eval(F,M), times(N,M,K).
  ```


Example (f(X,Y) :- eval(s(X*X) - X*X, Y).)

```
                                   □
                          Y = s(0) ‖
                plus(s(s(s(s(0)))),Y,s(s(s(s(s(0))))))
                  M = s(s(s(s(0)))) ‖
          eval(s(s(0))*s(s(0)),M), plus(M,Y,s(s(s(s(s(0))))))
                 N1 = s(s(s(s(0)))) ‖
   times(s(s(0)),s(s(0)),N1), eval(s(s(0))*s(s(0)),M), plus(M,Y,s(N1))
                       N3 = s(s(0)) ‖
eval(s(s(0)),N3), times(s(s(0)),N3,N1), eval(s(s(0))*s(s(0)),M), plus(M,Y,s(N1))
                            N5 = 0 |
eval(0,N5), eval(s(s(0)),N3), times(s(s(N5)),N3,N1), eval(s(s(0))*s(s(0)),M), plus(M,Y,s(N1))
                         N4 = s(N5) |
eval(s(0),N4), eval(s(s(0)),N3), times(s(N4),N3,N1), eval(s(s(0))*s(s(0)),M), plus(M,Y,s(N1))
                         N2 = s(N4) |
eval(s(s(0)),N2), eval(s(s(0)),N3), times(N2,N3,N1), eval(s(s(0))*s(s(0)),M), plus(M,Y,s(N1))
                                    |
       eval(s(s(0))*s(s(0)),N1), eval(s(s(0))*s(s(0)),M), plus(M,Y,s(N1))
                          N = s(N1) |
        eval(s(s(s(0))*s(s(0))),N), eval(s(s(0))*s(s(0)),M), plus(M,Y,N)
                                    |
              eval(s(s(s(0))*s(s(0))) - s(s(0))*s(s(0)),Y)
                                    |
                               f(s(s(0)),Y)
```


Speeding up evaluation using "let"

- consider sub-expression `X*X`
- solution: $f(x) = (let\ x2 = x^2\ in\ s(x2) - x2)$
- adding support for let in evaluator
- `let(X,E,F)` encodes *let x = e in f*

  ```
  eval(0,0).
  eval(s(E),s(N)) :- eval(E,N).
  eval(E+F,K) :- eval(E,N), eval(F,M), plus(N,M,K).
  eval(E-F,K) :- eval(E,N), eval(F,M), plus(M,K,N).
  eval(E*F,K) :- eval(E,N), eval(F,M), times(N,M,K).
  eval(let(X,E,F),K) :- eval(E,N), X = N, eval(F,K).
  ```

Example

```
f(X,Y) :- eval(s(X*X) - X*X, Y).
f(X,Y) :- eval(let(X2, X*X, s(X2) - X2), Y).
```

Example (f(X,Y) :- eval(let(X2,X*X,s(X2)-X2), Y).)

```
                              □
                    Y = s(0) ‖
          plus(s(s(s(s(0)))),Y,s(s(s(s(s(0))))))
               M = s(s(s(s(0)))) ‖
  eval(s(s(s(s(0)))),M), plus(M,Y,s(s(s(s(s(0))))))
             N = s(s(s(s(s(0))))) ‖
eval(s(s(s(s(s(0))))),N), eval(s(s(s(s(0)))),M), plus(M,Y,N)
                              |
          eval(s(s(s(s(s(0)))))-s(s(s(s(0)))),Y)
               X2 = s(s(s(s(0)))) |
           X2 = s(s(s(s(0)))), eval(s(X2)-X2,Y)
               N = s(s(s(s(0)))) ‖
  eval(s(s(0))*s(s(0)),N), X2 = N, eval(s(X2)-X2,Y)
                              |
        eval(let(X2,s(s(0))*s(s(0)),s(X2)-X2),Y)
                              |
                        f(s(s(0)),Y)
```


Speeding up "let" even further

- detected problems:
  1. after computing x^2, result is evaluated again `eval(s(s(s(s(0)))),M)`
  2. eval also steps into initial input
- solution: add new constructor *num* which states that the argument is a number, and hence, does not have to be evaluated

```
eval(0,0).
eval(s(E),s(N)) :- eval(E,N).
eval(E+F,K) :- eval(E,N), eval(F,M), plus(N,M,K).
eval(E-F,K) :- eval(E,N), eval(F,M), plus(M,K,N).
eval(E*F,K) :- eval(E,N), eval(F,M), times(N,M,K).
eval(num(N),N).
eval(let(X,E,F),K) :- eval(E,N),X = num(N), eval(F,K).
```


Example (f(X,Y):-GX=num(X),eval(let(X2,GX*GX,s(X2)-X2),Y))

```
                                        □
                              Y = s(0) ‖
                    plus(s(s(s(s(0)))),Y,s(s(s(s(s(0))))))
                       M = s(s(s(s(0)))) |
           eval(num(s(s(s(s(0))))),M), plus(M,Y,s(s(s(s(s(0))))))
                      N1 = s(s(s(s(0)))) |
eval(num(s(s(s(s(0))))),N1), eval(num(s(s(s(s(0))))),M), plus(M,Y,s(N1))
                                N = s(N1) |
 eval(s(num(s(s(s(s(0)))))),N), eval(num(s(s(s(s(0))))),M), plus(M,Y,N)
                                        |
              eval(s(num(s(s(s(s(0))))))-num(s(s(s(s(0))))),Y)
                   X2 = num(s(s(s(s(0))))) |
                 X2 = num(s(s(s(s(0))))), eval(s(X2)-X2,Y)
                        N = s(s(s(s(0)))) ‖
           times(s(s(0)),s(s(0)),N), X2 = num(N), eval(s(X2)-X2,Y)
                             N2 = s(s(0)) |
eval(num(s(s(0)),N2), times(s(s(0)),N2,N), X2 = num(N), eval(s(X2)-X2,Y)
                             N1 = s(s(0)) |
eval(num(s(s(0)),N1), eval(num(s(s(0)),N2), times(N1,N2,N), X2 = num(N), eval(s(X2)-X2,Y)
                                        |
       eval(num(s(s(0)))*num(s(s(0))),N), X2 = num(N), eval(s(X2)-X2,Y)
                                        |
              eval(let(X2,num(s(s(0)))*num(s(s(0))),s(X2)-X2),Y)
                         GX = num(s(s(0))) |
              GX = num(s(s(0))), eval(let(X2,GX*GX,s(X2)-X2),Y)
                                        |
                                  f(s(s(0)),Y)
```