Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016

Summary of Last Lecture

Example (meta-variable facility)

```
X; Y :- X.
X; Y :- Y.
```

Definitions (second-order programming)

- the predicate *bagof*(*Template*, *Goal*, *Bag*) unifies *Bag* with the alternatives of *Template* that meet *Goal*
- if *Goal* has free variables besides the one sharing with *Template* *bagof* will backtrack
- fails if *Goal* has no solutions
- construct *Var*^*Goal* tells *bagof* to existentially quantify *Var*
- the predicate *setof*(*Template*, *Goal*, *Bag*) is similar to *bagof* but sorts the obtained multi-set (bag) and removed duplicates

Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, logic foundations, unification, semantics, database and recursive programming, termination, complexity

Incomplete Data Structures and Constraints

incomplete data structures, definite clause grammars, constraint logic programming, answer set programming

Full Prolog

semantics (revisted), cuts, correctness proofs, meta-logical predicates, pragmatics, efficient programs, meta programming

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
- 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, if we argue wrt a linked-list implementation of cons lists?

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 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).
```


Excursion: Transforming Recursion into Iteration

Definitions

- a Prolog clause is called iterative if
  1. it has one recursive call, and
  2. zero or more calls to system predicates, before the recursive call
- a Prolog procedure is iterative if it contains only facts and iterative clauses

Example (Factorial Iterative, Version 1)

```
factorial(N,F) :- factorial(0,N,1,F).

factorial(I,N,T,F) :-
    I < N, I1 is I + 1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

Example (Factorial Iterative, Version 2)

```
factorial(N,F) :− factorial(N,1,F).

factorial(N,T,F) :−
    N > 0, T1 is T * N, N1 is N-1, factorial(N1,T1,F).
factorial(0,F,F).
```

Example

```
between(I,J,I) :− I ⩽ J.
between(I,J,K) :− I < J, I1 is I+1, between(I1,J,K).
```

Example

```
sumlist(Is,Sum) :− sumlist(Is,0,Sum).

sumlist([I|Is],Temp,Sum) :−
    Temp1 is Temp + I,sumlist(Is,Temp1,Sum).
sumlist([],Sum,Sum).
```


Example

```
maximum([X|Xs],M) :− maximum(Xs,X,M).

maximum([X|Xs],Y,M) :−
    X ⩽ Y, maximum(Xs,Y,M).
maximum([X|Xs],Y,M) :−
    X > Y, maximum(Xs,X,M).
maximum([],M,M).
```

Example

```
length([X|Xs],N) :−
    N >0, N1 is N - 1, length(Xs,N1).
length([],0).

length([X|Xs],N) :−
    length(Xs,N1), N is N1 + 1.
length([],0).
```


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.

Observation

- iterative programs are tail recursive
- sometimes tail recursion in general can be implemented as iteration which doesn't require a stack

Definition (tail recursion optimisation)

- consider a generic clause for A

$$A' :- B_1, \dots, B_n$$

  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; may require clause indexing

Definition

clause indexing is used to detect which clauses are applicable for reduction: 2nd clause in append need only be considered for empty lists

How to 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 specify relations and every function can be seen as a relation

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

- that is, we implement functions $f(i_1, \ldots, i_n) = (o_1, \ldots, o_m)$ by relations $f_{rel}/(n+m)$
- result is obtained by query $f_{rel}(i_1, \ldots, i_n, X_1, \ldots, X_m)$
  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 (actually a simple meta interpretor)

```
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)
```