Programming Paradigms

Summer Term 2017

14th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Logic programming: summary (1)

- principle of logic programming:
  - specification = collection of predicate definitions
  - predicate definition = sequence of clauses (facts and rules)
  - operationalisation =
    - essentially: step-wise resolution of (positive) literals
    - but also:
      - sequential (left-to-right) execution of conjunctions
      - backtracking (constrained by Cut-operator, which we haven't looked at)

- expressions/terms:
  - constants, variables
  - composite expressions: lists, structures (uninterpreted terms)
  - evaluable expressions: only for built-in arithmetic operators in **`is`**-literals
  - no nested predicate applications

- literals:
  - atomic formulas (with parameter list consisting of terms)
  - negated literals possible: **`not`**, **`\+`** , **`\=`**
  - literals with built-in predicates possible (e.g. **`is`**- or comparison literals)

Logic programming: summary (2)

- clauses:
  - facts: positive literals
  - rules:
    - head: positive literal
    - body: literal or conjunction of literals, possibly negative ones
  - recursion

- declarative semantics: motivated by logical model theory

- resolution/derivation trees:
  - unification as “two-way”-parameter passing, free variables, call modes
  - in special cases: analogous to pattern matching in Haskell
  - different clauses for same predicate are all explored (in top-down order), nondeterminism
  - operational impact of order of literals within a clause

- non-logical features:
  - negation as failure
  - some others (…)

Programming Paradigms

Prolog extension: DCGs

Symbolic language processing/representation (1)

- Assume we want to model sentences of the English language.
- We need different categories of words and sentence parts:

  verb, noun, verb phrase, …

  as well as rules for grammatically correct combination of those:

sentence	→	noun phrase, verb phrase
noun phrase	→	determiner, noun
verb phrase	→	verb, noun phrase
	…	

- And, of course, a mechanism for “executing” such a grammar.

Symbolic language processing/representation (2)

Simple realization in Prolog:

- Word categories + rules:

```
det([the]).
det([a]).

n([woman]).
n([man]).

v([knows]).
```

```
np(Z) :- det(X), n(Y), append(X,Y,Z).

vp(Z) :- v(X), np(Y), append(X,Y,Z).
vp(Z) :- v(Z).

s(Z) :- np(X), vp(Y), append(X,Y,Z).
```

- Usage:

```
?- s([a,woman,knows,a,man]).
true.

?- s([the,woman,knows]).
true.

?- s(Z).
Z = [the, woman, knows, the, woman] ;
…
Z = [a, man, knows].
```

Somewhat nice, but potentially quite inefficient due to the way of using `append`!

Symbolic language processing/representation (3)

Special Prolog feature: “Definite Clause Grammars”

```
det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [knows].
```

```
np --> det, n.

vp --> v, np.
vp --> v.

s --> np, vp.
```

Usage (with special role of second argument, instantiated with empty list):

```
?- s([a,woman,knows,a,man],[]).
true.

?- s([the,woman,knows],[]).
true.

?- s(Z,[]).
Z = [the, woman, knows, the, woman] ;
…
Z = [a, man, knows].
```

Symbolic language processing/representation (4)

So far we can only test or generate:

```
?- s([a,woman,knows,a,man],[]).
true.

?- s(Z,[]).
Z = [the, woman, knows, the, woman] ;
…
Z = [a, man, knows].
```

In addition, we would like to truly “parse”, that is, with output of sentence structure.

By adding a syntax tree argument:

```
det(td) --> [the].
det(td) --> [a].

n(tn) --> [woman].
n(tn) --> [man].

v(tv) --> [knows].
```

```
np(tnp(T,S)) --> det(T), n(S).

vp(tvp(T,S)) --> v(T), np(S).
vp(tvp(T))   --> v(T).

s(ts(T,S)) --> np(T), vp(S).
```

Symbolic language processing/representation (5)

```
det(td) --> [the].
det(td) --> [a].

n(tn) --> [woman].
n(tn) --> [man].

v(tv) --> [knows].
```

```
np(tnp(T,S)) --> det(T), n(S).

vp(tvp(T,S)) --> v(T), np(S).
vp(tvp(T))   --> v(T).

s(ts(T,S)) --> np(T), vp(S).
```

```
?- s(T,[a,woman,knows,a,man],[]).
T = ts(tnp(td,tn),tvp(tv,tnp(td,tn))).

?- s(T,Z,[]).
T = ts(tnp(td,tn),tvp(tv,tnp(td,tn))),
Z = [the, woman, knows, the, woman] ;
…
T = ts(tnp(td,tn),tvp(tv)),
Z = [a, man, knows].
```

Symbolic language processing/representation (6)

Another sensible use of additional arguments: grammatical features.

- Assume we want to introduce pronouns:

```
det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [knows].

pro --> [he].
pro --> [she].
pro --> [him].
pro --> [her].
```

```
np --> pro.
np --> det, n.

vp --> v, np.
vp --> v.

s --> np, vp.
```

- Hmm:

```
?- s(Z,[]).
Z = [he, knows, he] ;
Z = [he, knows, she] ; …
```

Symbolic language processing/representation (7)

- Corrections by way of additional arguments:

```
det --> [the].
det --> [a].

n --> [woman].
n --> [man].

v --> [knows].

pro(subject) --> [he].
pro(subject) --> [she].
pro(object) --> [him].
pro(object) --> [her].
```

```
np(X) --> pro(X).
np(_) --> det, n.

vp --> v, np(object).
vp --> v.

s --> np(subject), vp.
```

- Now:

```
?- s(Z,[]).
Z = [he, knows, him] ;
Z = [he, knows, her] ;
Z = [he, knows, the, woman] ;
Z = [he, knows, the, man] ;
Z = [he, knows, a, woman] ; …
```

Another example: parsing of arithmetic expressions

- As a reminder:

```
⟨Expr⟩   ::= ⟨Term⟩ ‘+’ ⟨Expr⟩ | ⟨Term⟩
⟨Term⟩   ::= ⟨Factor⟩ ‘*’ ⟨Term⟩ | ⟨Factor⟩
⟨Factor⟩ ::= ⟨Nat⟩ | ‘(’ ⟨Expr⟩ ‘)’
```

- Realization in Haskell (but not further explained in the lecture):

```
expr   =  ( ADD <$> term <* char '+' <*> expr ) ||| term

term   =  ( MUL <$> factor <* char '*' <*> term ) ||| factor

factor =  ( LIT <$> nat ) ||| ( char '(' *> expr <* char ')' )
```

Another example: parsing of arithmetic expressions

- Now in Prolog:

```
expr(+(T,E)) --> term(T),"+",expr(E).
expr(T)      --> term(T).

term(*(F,T)) --> factor(F),"*",term(T).
term(F)      --> factor(F).

factor(N) --> nat(N).
factor(E) --> "(",expr(E),")".

nat(0) --> "0".
…
nat(9) --> "9".
```

- Tests:

```
?- expr(E,"1+2*3",""), R is E.
E = 1+2*3, R = 7.

?- expr((1+2)*3,S,"").
S = [40, 49, 43, 50, 41, 42, 51] ;

?- expr((1+2)*3,S,""), writef("%s",[S]).
(1+2)*3
```

Another example: parsing of arithmetic expressions

- Exploiting different call modes:

```
parse(S,E) :- expr(E,S,"").

pretty_print(E,S) :- expr(E,S,"").

normalize(S,T) :- parse(S,E),pretty_print(E,T).
```

- Tests:

```
?- parse("1+(2*3)",E), R is E.
E = 1+2*3, R = 7.

?- pretty_print(1+2*3,S), !, writef("%s",[S]).
1+2*3

?- normalize("1+(2*3)",S), !, writef("%s",[S]).
1+2*3

?- normalize("(1+2)*3",S), !, writef("%s",[S]).
(1+2)*3
```

Programming Paradigms

Prolog extension: dynamic predicates

As a reminder: transitive closure, but now with a cycle

```
direct(frankfurt,san_francisco).
direct(frankfurt,chicago).
direct(san_francisco,honolulu).
direct(honolulu,maui).
direct(honolulu,san_francisco).

connection(X, Y) :- direct(X, Y).
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
Y = honolulu ;
Y = maui ; …
```

Aim should be: avoid infinite search

As a reminder: transitive closure, but now with a cycle

Idea: remember already visited stations, for example as a list:

```
direct(frankfurt,san_francisco).
…
direct(honolulu,san_francisco).

connection(X, Y) :- connection1(X, Y, [X]).

connection1(X, Y, _) :- direct(X, Y).
connection1(X, Y, L) :- direct(X, Z), not(member(Z,L)),
                        connection1(Z, Y, [Z|L]).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.
```

Cumbersome. And maybe too inefficient: linear search in that stations list.

Program facts as data structure

Alternative: save the visited stations as Prolog program facts.

```
direct(frankfurt,san_francisco).
…
direct(honolulu,san_francisco).

connection(X, Y) :- assert(visited(X)), connection2(X, Y).

connection2(X, Y) :- direct(X, Y).
connection2(X, Y) :- direct(X, Z), not(visited(Z)),
                     assert(visited(Z)), connection2(Z, Y).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.

?- connection(san_francisco,Y).
Y = honolulu ;
false.
```

Oops!

Program facts as data structure

Cleaning up:

```
direct(frankfurt,san_francisco).
...
direct(honolulu,san_francisco).

connection(X, Y) :- retractall(visited(_)),
                    assert(visited(X)), connection2(X, Y).

connection2(X, Y) :- direct(X, Y).
connection2(X, Y) :- direct(X, Z), not(visited(Z)),
                     assert(visited(Z)), connection2(Z, Y).
```

```
?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.

?- connection(san_francisco,Y).
Y = honolulu ;
Y = maui ;
Y = san_francisco ;
false.
```

Program facts as data structure

Example uses of the meta predicates **`assert`** and **`retract`**:

```
1 ?- listing.
true.

2 ?- assert(p(1)).
true.

3 ?- assert(p(1)).
true.

4 ?- assert(p(2)).
true.

5 ?- listing.

:- dynamic p/1.
p(1).
p(1).
p(2).
true.
```

```
6 ?- p(X).
X = 1 ;
X = 1 ;
X = 2.

7 ?- retract(p(1)).
true.

8 ?- p(X).
X = 1 ;
X = 2.

9 ?- retract(p(X)).
X = 1 ;
X = 2.

10 ?- listing.

:- dynamic p/1.
true.
```

Program facts as data structure

- Another useful application of **`assert`** is memoization.

- As a reminder, in Haskell (unmemoized):

```
fib 0  =  1
fib 1  =  1
fib n  =  fib (n – 1) + fib (n – 2)
```

```
fib(N,1) :- N<2, !.
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2.
```

- The problem:

```
?- fib(10,X).
X = 89.

?- fib(30,X).
X = 1346269.

?- fib(50,X).
```

hopeless

Program facts as data structure

```
fib(N,1) :- N<2, !.
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2.
```

⬇

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

fib(N,1) :- N<2, !.
fib(N,M) :- memo(N,M), !.
fib(N,M) :- N1 is N-1, fib(N1,M1), N2 is N-2, fib(N2,M2), M is M1+M2,
            assert(memo(N,M)).
```

- Now:

```
?- fib(10,X).
X = 89.

?- fib(30,X).
X = 1346269.

?- fib(50,X).
X = 20365011074.
```

instantaneous

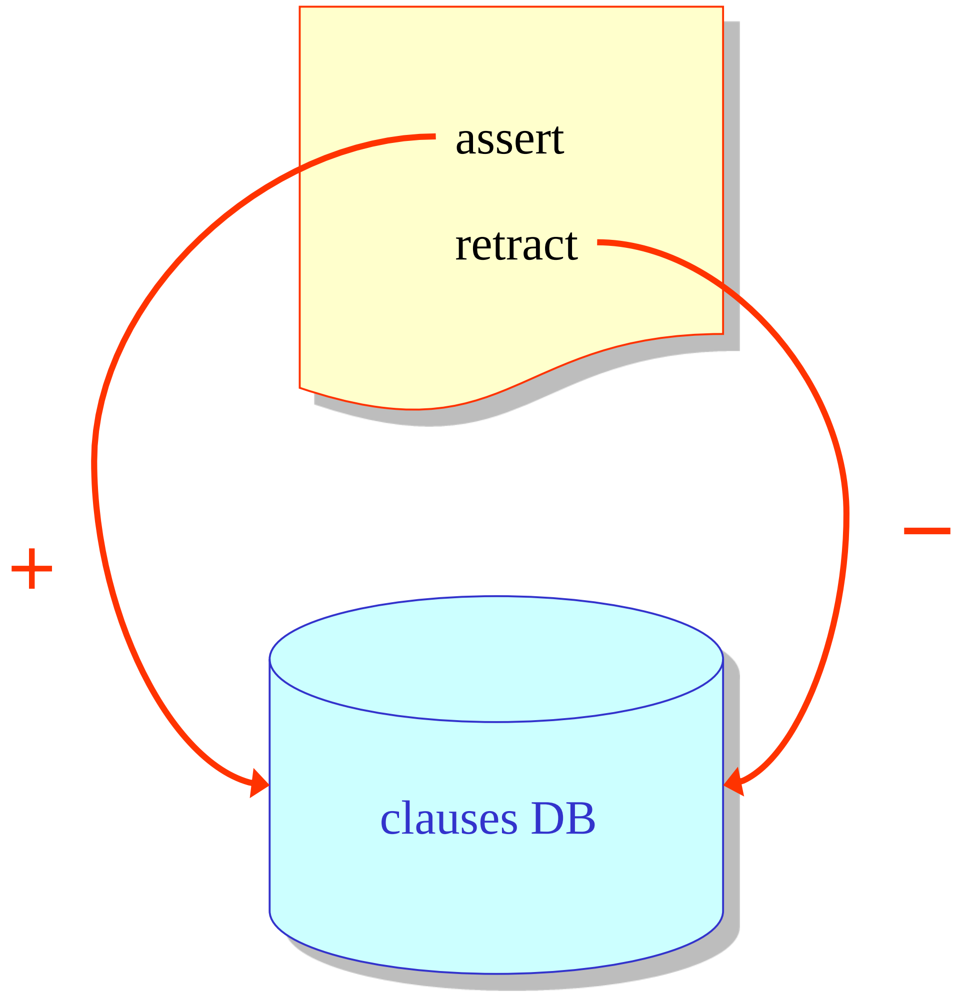
Program facts as data structure

Side effects on the
“data base” of clauses!

Two variants are common:

1) “DB” as additional
data structure (facts)
⇒ (almost) normal in Prolog

2) self modification of the
program
(“DB” as program)
⇒ meta programming



Programming Paradigms

Prolog extension: collection predicates

Generating all solutions to a query (1)

- Often several solutions to a query exist:

```
child(martha, charlotte).
child(charlotte, caroline).
child(caroline, laura).
child(laura, rose).

descend(X, Y) :- child(X, Y).
descend(X, Y) :- child(X, Z), descend(Z, Y).
```

The query `?- descend(martha,X).` would successively yield the answers `X = charlotte`, `X = caroline`, `X = laura` and `X = rose`.

- Prolog offers three different meta predicates for generating all solutions “in one go”:

`findall` , `bagof` , `setof`

in each case delivering them in a result list in a certain way.

Generating all solutions to a query (2)

```
findall(Template, Goal, List).
```

- For every solution of the query `Goal`, the instantiated `Template` is included in the result list `List`.

```
?- findall(X, descend(martha, X), Z).
Z = [charlotte, caroline, laura, rose].
```

- The term `Template` can be a complex structure with (or without) variables, from which the entries of the result list are then built.

```
?- findall(fromMartha(X), descend(martha, X), Z).
Z = [fromMartha(charlotte), fromMartha(caroline),
                 fromMartha(laura), fromMartha(rose)].
```

Generating all solutions to a query (3)

Variants **`bagof`** and **`setof`** behave slightly differently (concerning binding of variables, and concerning duplicates and sorting).

Possible application of the collection predicates: simulation of list comprehensions.

Haskell: $[\, e \mid x \leftarrow xs \,]$

↓

Prolog:

```
findall(E, member(X,Xs), List).
```

Generating all solutions to a query (4)

Examples:

Prolog equivalents for the following Haskell definitions?

1. [n .. m]
2. [n, m .. l]
3. [x * x | x ← [1 .. 100], x `mod` 2 == 0]

Possible solutions for 1.:

```
fromTo(N,M,L)      :- N > M, !, L = [].
fromTo(N,M,[N|L])  :- N1 is N+1, fromTo(N1,M,L).
```

or

```
fromTo(N,M,L)  :- findall(X,between(N,M,X),L).
```

Generating all solutions to a query (5)

Examples:

Prolog equivalents for the following Haskell definitions?

2. [n, m .. l]

3. [x * x | x ← [1 .. 100], x `mod` 2 == 0]

Possible solutions for 2.:

```
fromThenTo(N,M,L,Xs)     :- (N >= M; N > L), !, Xs = [].
fromThenTo(N,M,L,[N|R]) :- M1 is M+M-N, fromThenTo(M,M1,L,R).
```

or

```
fromThenTo(N,M,L,Xs) :- (N >= M; N > L), !, Xs = [].
fromThenTo(N,M,L,Xs) :- D is M-N, fromTo(0,(L-N)/D,Is),
                        findall(X,(member(I,Is), X is N+I*D),Xs).
```

Generating all solutions to a query (6)

Examples:

Prolog equivalents for the following Haskell definitions?

3. [x * x | x ← [1 .. 100], x `mod` 2 == 0]

Possible solutions for 3.:

```
squares(L) :- fromTo(1,100,Xs), filter(Xs,Ys), map(Ys,L).

filter([],[]).
filter([X|Xs],[X|Ys]) :- X mod 2 =:= 0, !, filter(Xs,Ys).
filter([_|Xs],Ys)     :- filter(Xs,Ys).

map([],[]).
map([X|Xs],[Y|Ys]) :- Y is X*X, map(Xs,Ys).
```

or

```
squares(L) :- fromTo(1,100,Xs),
              findall(Y,(member(X,Xs), X mod 2 =:= 0, Y is X*X),L).
```

Programming Paradigms

FP vs. LP (or not so much "vs."?)

FP vs. LP: some general correspondences

functional (Haskell)	logic (Prolog)
function	relation / predicate
equation	clause
nesting of expressions	conjunction of literals
reduction	resolution
pattern matching	unification
lazy evaluation (leftmost-outermost)	sequential processing (left-right)
list comprehensions	findall / bagof / setof
parser combinators	definite clause grammars

FP vs. LP: some general differences

functional (Haskell)	logic (Prolog)
???	free variables, call modes
???	solution alternatives
???	backtracking
???	negation
types, polymorphism	???
higher-order	???
mathematical purity	(to some extent)

Functional-logic programming

For example in the language Curry:

```
coin :: Int
coin = 0
coin = 1
```

```
double :: Int → Int
double x = x + x
```

```
> coin
0
More?
1
More?
No more Solutions
```

```
> double  coin
0
More?
2
More?
No more Solutions
```

```
coin :: Int
coin = 0 ? 1
```

Functional-logic programming

For example in the language Curry:

```
f :: a → [a] → [a]
f x ys       =  x : ys
f x (y : ys) =  y : f x ys
```

```
g :: [a] → [a]
g [ ]        =  [ ]
g (x : xs)   =  f x (g xs)
```

```
> f 3 [1, 2]
[1, 2, 3]
More?
[1, 3, 2]
More?
[3, 1, 2]
More?
No more Solutions
```

```
> g [1, 2, 3]
[3, 2, 1]
More?
[3, 1, 2]
More?
[2, 3, 1]
More?
...
```

Functional-logic programming

For example in the language Curry:

```
list :: [Int]
list = ys ++ [1]
    where  ys free
```

```
f :: [a] → a
f xs | ys ++ [y] == xs  =  y
    where  ys, y free
```

```
> list
[1]
More?
[_a, 1]
More?
[_a, _b, 1]
More?
…
```

```
> f  [1 .. 4]
4
More?
No more Solutions
```

```
f :: [a] → a
f (_ ++ [y]) =  y
```

Zebra puzzle functional-logically (1)

```
data Color        =  Red | Yellow | Blue | Green | Ivory
data Nationality  =  Norwegian | Englishman | Spaniard | Ukrainian | Japanese
data Drink        =  Coffee | Tea | Milk | Juice | Water
data Pet          =  Dog | Horse | Snails | Fox | Zebra
data Smoke        =  Winston | Kools | Chesterfield | Lucky | Parliaments

right_of :: a → a → [a] → Success
right_of r l (h1 : h2 : hs)  = (l =:= h1 & r =:= h2)  ?  right_of r l (h2 : hs)

next_to :: a → a → [a] → Success
next_to x y = right_of  x y
next_to x y = right_of  y x

member :: a → [a] → Success
member x (y : ys)  = x =:= y  ?  member x ys
```

Zebra puzzle functional-logically (2)

```
zebra :: ([(Color,Nationality,Drink,Pet,Smoke)], Nationality)
zebra  |  member (Red, Englishman, _, _, _) houses
          & member (_, Spaniard, _, Dog, _) houses
          & member (Green, _, Coffee, _, _) houses
          & member (_, Ukrainian, Tea, _, _) houses
          & right_of (Green, _, _, _, _) (Ivory, _, _, _, _) houses
          & member (_, _, _, Snails, Winston) houses
          & member (Yellow, _, _, _, Kools) houses
          & next_to (_, _, _, _, Chesterfield) (_, _, _, Fox, _) houses
          & next_to (_, _, _, _, Kools) (_, _, _, Horse, _) houses
          & member (_, _, Juice, _, Lucky) houses
          & member (_, Japanese, _, _, Parliaments) houses
          & next_to (_, Norwegian, _, _, _) (Blue, _, _, _, _) houses
          & member (_, zebraOwner, _, Zebra, _) houses
          & member (_, _, Water, _, _) houses
       = (houses, zebraOwner)
       where
          houses = [(_, Norwegian, _, _, _), _, (_, _, Milk, _, _), _, _]
          zebraOwner = _
```