Computational Logic

Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016

Summary of Last Lecture

Example (design as function)

```
delete([X|Xs],X,Ys) :-
        delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :-
        dif(X,Z),
        delete(Xs,Z,Ys).
delete([],_X,[]).
```

Example (use as relation)

```
delete2([X|Xs],X,Ys) :-
        delete2(Xs,X,Ys).
delete2([X|Xs],Z,[X|Ys]) :-
        delete2(Xs,Z,Ys).
delete2([],_X,[]).
```


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), correctness proofs, meta-logical predicates, cuts nondeterministic programming, efficient programs, complexity

SWI-Prolog

```
[zid-gpl.uibk.ac.at] swipl
  Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.2.3)
  Copyright (c) 1990-2009 University of Amsterdam.
  SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
  and you are welcome to redistribute it under certain conditions.
  Please visit http://www.swi-prolog.org for details.

  For help, use ?- help(Topic). or ?- apropos(Word).

  ?-
```

SWI-Prolog Emacs Mode

Bruda's Prolog Mode

1. goto `http://bruda.ca/emacs/prolog_mode_for_emacs`
2. download prolog.el, compile and put into sub-directory site-lisp
3. put the following into .emacs:

```
(autoload 'run-prolog "prolog"
          "Start a Prolog sub-process." t)
(autoload 'prolog-mode "prolog"
          "Major mode for editing Prolog programs." t)
(setq prolog-system 'swi)
(setq auto-mode-alist
      (cons (cons "\\.pl" 'prolog-mode) auto-mode-alist))
```


Example (Xs is a subset of Ys)

```
members([X|Xs],Ys) :- member(X,Ys), members(Xs,Ys).
members([],Ys).
```

Example (Xs is a subset of Ys)

```
selects([X|Xs],Ys) :- select(X,Ys,Ys1), selects(Xs,Ys1).
selects([],Ys).
```

Observations

1. *members/2* ignores the multiplicity of elements
2. *members/2* terminates iff 1st argument is complete
3. the first restriction is lifted, the second altered with *selects/2*
4. *selects/2* strongly normalises iff 2nd argument is complete; weakly normalises iff at least one argument is complete

Example

```
%    no_doubles(Xs,Ys) <--
%       Ys is the list obtained by removing duplicate
%       elements from the list Xs
```

Example

```
non_member(X,[Y|Ys]) :- dif(X,Y), non_member(X,Ys).
non_member(X,[]).

no_doubles([X|Xs],Ys) :-
    member(X,Xs), no_doubles(Xs,Ys).
no_doubles([X|Xs],[X|Ys]) :-
    non_member(X,Xs), no_doubles(Xs,Ys).
no_doubles([],[]).
```


Built-in Predicates for List Manipulation

- `append/3`
- `member/2`
- `last/2`

```
?- last([a,b,c,d],X).
X = d
```

```
?- last(X,a).
X = [a] ;
X = [_G324,a] ;
X = [_G324,_G327,a]
```

- `reverse/2`

```
?- reverse([a,b,c,d],X).
X = [d,c,b,a]
```

- `select/3`

```
?- select(b,[a,b,c,d],X).
X = [a,c,d]
```

```
?- select(b,[a,b,c,b,d],X).
X = [a,c,b,d]
```

- `length/2`

```
?- length([a,b,c,d],X).
X = 4
```

Incomplete Data Structures

Observation

given a list [1,2,3] it can be represented as the difference of two lists

1. [1,2,3] = [1,2,3] \ []
2. [1,2,3] = [1,2,3,4,5] \ [4,5]
3. [1,2,3] = [1,2,3,8] \ [8]
4. [1,2,3] = [1,2,3|Xs] \ Xs

Definition

the difference of two lists is denotes as *As* \ *Bs* and called difference list

Example

```
append_dl(Xs \ Ys, Ys \ Zs, Xs \ Zs).
```


Application of Difference Lists

Example

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

Example

```
quicksort(Xs,Ys) :- quicksort_dl(Xs, Ys \ []).
quicksort_dl([X|Xs], Ys \ Zs) :-
    partition(Xs,X,Littles, Bigs),
    quicksort_dl(Littles,Ys \ [X|Ys1]),
    quicksort_dl(Bigs,Ys1 \ Zs).
quicksort_dl([],Xs \ Xs).
```


Observations

- difference lists are effective if independently different sections of a list are built, which are then concatenated
- the separation operator \ simplifies reading, but can be eliminated: “As \ Bs” → “As , Bs”
- the explicit constructor should be removed, if time or space efficiency is an issue

More Observations

- the tail *Bs* of a difference list acts like a pointer to the end of the first list *As*
- this works as *As* is an incomplete list
- thus we represent a concrete list as the difference of two incomplete data structures
- generalises to other recursive data types

Difference-structures

Example

convert the sum $(a + b) + (c + d)$ into $(a + (b + (c + (d + 0))))$

Definition

we make use of difference-sums: $E1{+}{+}E2$, where $E1$, $E2$ are incomplete; the empty sum is denoted by 0

Example

```
normalise(Exp,Norm) :- normalise_ds(Exp,Norm ++ 0).
normalise_ds(A+B, Norm ++ Norm0) :-
    normalise_ds(A, Norm ++ NormB),
    normalise_ds(B, NormB ++ Norm0).
normalise_ds(A,(A + Norm) ++ Norm) :-
    constant(A).
```

Context-Free Grammars

Definition

a grammar G is a tuple $G = (V, \Sigma, R, S)$, where

1. V finite set of variables (or nonterminals)
2. Σ alphabet, the terminal symbols, $V \cap \Sigma = \varnothing$
3. R finite set of rules
4. $S \in \mathcal{V}$ the start symbol of G

a rule is a pair $P \to Q$ of words, such that $P, Q \in (V \cup \Sigma)^*$ and there is at least one variable in P

Definition

grammar $G = (V, \Sigma, R, S)$ is context-free, if $\forall$ rules $P \to Q$:

1. $P \in V$
2. $Q \in (V \cup \Sigma)^*$

Example

```
sentence → noun_phrase, verb_phrase.
noun_phrase → determiner, noun_phrase2.
noun_phrase → noun_phrase2.
noun_phrase2 → adjective, noun_phrase2.
noun_phrase2 → noun.
verb_phrase → verb, noun_phrase.
verb_phrase → verb.
determiner → [the].
determiner → [a].
noun → [pie-plate].
noun → [surprise].
adjective → [decorated].
verb → [contains].
```

sentence $\overset{*}{\Rightarrow}$ ‘‘the decorated pie-plate contains a surprise’’

Example

```
sentence(S \ S0) :- noun_phrase(S \ S1), verb_phrase(S1 \ S0).
noun_phrase(S \ S0) :-
    determiner(S \ S1), noun_phrase2(S1 \ S0).
noun_phrase(S) :- noun_phrase2(S).
noun_phrase2(S \ S0) :-
    adjective(S \ S1), noun_phrase2(S1 \ S0).
noun_phrase2(S) :- noun(S).
verb_phrase(S \ S0) :- verb(S \ S1), noun_phrase(S1 \ S0).
verb_phrase(S) :- verb(S).
determiner([the|S] \ S).
determiner([a|S] \ S).
noun([pie-plate|S] \ S).
noun([surprise|S] \ S).
adjective([decorated|S] \ S).
verb([contains|S] \ S).
```


Extension: Add Parsetree

Example

```
sentence(sentence(N,V), S \ S0) :-
    noun_phrase(N, S \ S1),
    verb_phrase(V, S1 \ S0).
```

Example (Definite Clause Grammars)

```
sentence(sentence(N,V)) → noun_phrase(N), verb_phrase(V).
noun_phrase(np(D,N)) → determiner(D), noun_phrase2(N).
noun_phrase(np(N)) → noun_phrase2(N).
noun_phrase2(np2(A,N)) → adjective(A), noun_phrase2(N).
noun_phrase2(np2(N)) → noun(N).
verb_phrase(vp(V,N)) → verb(V), noun_phrase(N).
verb_phrase(vp(V)) → verb(V).
```