Computational Logic

Logic Programming

Cezary Kaliszyk **Georg Moser**

Institute of Computer Science @ UIBK

Winter 2015

Termination Analysis

Fact

- *for termination analysis only recursive calls (cycles in call tree) are essential*
- *let's remove non-recursive rules*

Example (specialised)

```
ancestor_of_2(Ancestor, Predecessor) :- false,
    child_of(Predecessor, Ancestor),
ancestor_of_2(Ancestor, Predecessor) :-
    ancestor_of_2(Person, Predecessor),
    child_of(Person, Ancestor).
```

equivalently

```
ancestor_of_2(Ancestor, Predecessor) :-
    ancestor_of_2(Person, Predecessor),
    child_of(Person, Ancestor).
```


Summary of Last Lecture

Definitions

- a type is a (possible infinite) set of terms
- types are conveniently defined by unary relations
- a type is complete if closed under the instance relation
- with every complete type T one associates an incomplete type IT which is a set of terms with instances in T and instances not in T

Example (composition of programs)

```
ancestor_of_2(Ancestor, Predecessor) :-
    child_of(Predecessor, Ancestor),
ancestor_of_2(Ancestor, Predecessor) :-
    ancestor_of_2(Person, Predecessor),
    child_of(Person, Ancestor).
```


Outline of the Lecture

Monotone Logic Programs

introduction, basic constructs, unification, database and recursive programming, termination

Incomplete Data Structures and Constraints

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

Full Prolog

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

Composition of Programs

five steps to implement relation R

1. look up existing definitions of relation R
   - family relations
   - train tables
2. define types of individual data
   - is_number
   - mainly for documentation
3. think up a suitable name
   - convert a verbose description into a name
   - child_of
4. write assertions (use cases)
5. write the actual program

Negative Definitions

Definition

negative definitions define a relation with the help of negation

Example

```
land(X) ← not sea(X).
```

Fact

negative definitions are dangerous as their scope is usually larger than expected and they are difficult to maintain

Example

```
← land(27).
← land(kar_rinne).
← land(milka_kuh).
```


Structured Data and Data Abstraction

Example (Unstructured Data)

```
course(discrete_mathematics,tuesday,8,11,sandor,szedmak,
  victor_franz_hess,d).
```

Example (Structured Data)

```
course(discrete_mathematics,time(tuesday,8,11),
  lecturer(sandor,szedmak),location(victor_franz_hess,d)).
```

Example

```
lecturer(Lecturer,Course) :-
  course(Course,Time,Lecturer,Location).
duration(Course,Length) :-
  course(Course,time(Day,Start,Finish),Lecturer,Location),
  plus(Start,Length,Finish).
```


Example (cont'd)

```
teaches(Lecturer,Day) :-
  course(Course,time(Day,Start,Finish),Lecturer,Location).
occupied(Location,Day,Time) :-
  course(Course,time(Day,Start,Finish),Lecturer,Location),
  Start ⩽ Time, Time ⩽ Finish.
```

Why structure Data?

- helps to organise data
- rules can be written abstractly, hiding irrelevant detail
- modularity is improved

The Art of Prolog says

We believe that the appearance of a program is important, particularly when attempting difficult problems

Logic Programs and the Relational Database Model

Observation

the basic operations of relational algebras, namely:

1. union
2. difference
3. cartesian product
4. projection
5. selection
6. intersection

can easily be expressed within logic programming

Example

```
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).
```


Example (Uses of append)

```
prefix(Xs,Ys) :- append(Xs,As,Ys).
suffix(Xs,Ys) :- append(As,Xs,Ys).
member(X,Ys) :- append(As,[X|Xs],Ys).
```

Example

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

Example

```
length([],0).
length([X|Xs],s(N)) :- length(Xs,N).
```


Example

```
permutationsort(Xs,Ys) :- permutation(Xs,Ys), ordered(Ys).
permutation(Xs,[Z|Zs]) :- select(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).
ordered([X]).
ordered([X,Y|Ys]) :- X ⩽ Y, ordered([Y|Ys]).
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

Example

```
insertionsort([X|Xs],Ys) :- insertionsort(Xs,Zs),
                            insert(X,Zs,Ys).
insertionsort([],[]).
insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]) :- X > Y, insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :- X ⩽ Y.
```


Example

```
quicksort([X|Xs],Ys) :-
    partition(Xs,X,Littles,Bigs),
    quicksort(Littles,Ls), quicksort(Bigs,Rs),
    append(Ls,[X|Rs],Ys).

partition([X|Xs],Y,[X|Ls],Bs) :-
    X =< Y, partition(Xs,Y,Ls,Bs).
partition([X|Xs],Y,Ls,[X|Bs]) :-
    X > Y,     partition(Xs,Y,Ls,Bs).
partition([],Y,[],[]).
```

Example (Recursive Datastructures)

```
isotree(nil,nil).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Right2), isotree(Right1,Left2).
```

Accessing compound terms

Definition

- `functor`(*Term*,*F*,*Arity*) is true, if *Term* is a compound term, whose principal functor is *F* with arith *Arity*
- `arg`(*N*,*Term*,*Arg*) is true, if *Arg* is the N^{th} argument of *Term*

Example

```
:- functor(father(haran,lot),F,A)
```

```
F ↦ father
A ↦ 2
```

Example

```
:- arg(2,father(haran,lot),X)
```

```
X ↦ lot
```


Example

```
subterm(Term,Term).
subterm(Sub,Term) :-
    compound(Term),
    functor(Term,F,N),
    subterm(N,Sub,Term).

subterm(N,Sub,Term) :-
    N > 1,
    N1 is N - 1,
    subterm(N1,Sub,Term).
subterm(N,Sub,Term) :-
    arg(N,Term,Arg),
    subterm(Sub,Arg).
```


Definition

- *Term* `=..` *List* is true if *List* is a list whose head is the principal functor of *Term*, and whose tail is the list of arguments of *Term*
- the operator `=..` is also called `univ`

Example

```
:- father(haran,lot) =.. Xs
```

```
X ↦ [father,haran,lot]
```

Remark

- programs written with `functor` and `arg` can also be written with `univ`
- programs using `univ` are typically simpler
- programs using `functor` and `arg` are more efficient
- `univ` can be built from `functor` and `arg`

Approach

1. sometimes it is useful (easier) to think of a relation as a function
2. use this definition for coding
3. afterwards see, if alternative uses make declarative sense

Example

delete/3 removes all occurrences of an element from a list

Example

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

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

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

:- delete2([a,b,c,b],b,[a,c])
true
:- delete2([a,b,c,b],b,[a,b,c,b])
true
```

Example

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

:- delete([a],b,[a])
true
:- select([a],b,X)
false
```


GUPU

Example (termination)

- Example 29
- Example 30

Example (definite clause grammars)

- Example 37