Computational Logic

Logic Programming

Cezary Kaliszyk **Georg Moser**

Institute of Computer Science @ UIBK

Winter 2015

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

Summary of Last Lecture

Example

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Descendant, Ancestor).
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).
```

Example

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


Example (revisited)

```
ancestor_of(Ancestor, Descendant) :-
    child_of(Descendant, Ancestor).
ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).
```

- Ancestor
- Descendant
- Person

In English

> *Someone is ancestor of a descendant, if the descendant is his (or her) child, or if he (or she) has a child and this person is the ancestor of the descendant.*

binding of logical variables is expressed as references

Declarative Reading

Definition

the declarative reading of a program is its concept as (set of) logical formulas

Analysis

1. specialisation
   - if we remove clauses of a defined relation, then this relation becomes smaller; the program is specialised
   - if the specialisation provides wrong answers, the original program certainly will
2. generalisation
   - if we remove goals from the body of a clause, the relation is extended; the program is generalised
   - if the generalised program cannot derive correct facts, the original can neither

Procedure Reading

Example (multiplication)

logic program

```
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

ground queries

`:- plus(s(s(0)),s(0),s(s(s(0))))`	$X \mapsto s(0),\ Y \mapsto s(0),\ Z \mapsto s(s(0))$
`:- plus(s(0),s(0),s(s(0)))`	$X \mapsto 0,\ Y \mapsto s(0),\ Z \mapsto s(0)$
`:- plus(0,s(0),s(0))`	$X \mapsto s(0)$

solved

. . . is Too Complicated

Example (renaming is needed)

logic program

```
plus(0,X3,X3).
plus(s(X2),Y2,s(Z2)) :- plus(X2,Y2,Z2).
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

query

`:- plus(s(s(0)),s(0),X)`	$X_1 \mapsto s(0),\ Y_1 \mapsto s(0),\ X \mapsto s(Z_1)$
`:- plus(s(0),s(0),Z1)`	$X_2 \mapsto 0,\ Y_2 \mapsto s(0),\ Z_1 \mapsto s(Z_2)$
`:- plus(0,s(0),Z2)`	$X_3 \mapsto s(0),\ Z_2 \mapsto s(0)$

solution $X \mapsto s(s(s(0)))$

Definition

- a type is a (possible infinite) set of terms
- types are conveniently defined by unary relations

Example

```
male(X).    female(X).
```

Definition

- to define complex types, recursive logic programs may be necessary
- the latter types are called recursive types
- recursive types, defined by unary recursive programs, are called simple recursive types
- a program defining a type is a type definition; a call to a predicate defining a type is a type condition

Simple Recursive Types

Example

```
is_tree(nil).
is_tree(tree(Element,Left,Right)) :-
    is_tree(Left),
    is_tree(Right).
```

Definition

- 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

- the type $\{0, s(0), s(s(0)), \dots\}$ is complete
- the type $\{X, 0, s(0), s(s(0)), \dots\}$ is incomplete

Arithmetic

Example

```
is_number(0).
is_number(s(X)) :- is_number(X).
```

Example

```
plus(0,X,X) :- is_number(X)..
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
times(0,X,0).
times(s(X),Y,Z) :- times(X,Y,U), plus(U,Y,Z).
```

Example

```
factorial(0,s(0)).
factorial(s(N),F) :- factorial(N,F1), times(s(N),F1,F).
```


Example

```
0 ⩽ X :- is_number(X).
s(X) ⩽ s(Y) :- X ⩽ Y.
minimum(N1,N2,N1) :- N1 ⩽ N2.
minimum(N1,N2,N2) :- N2 ⩽ N1.
```

Example

```
mod(X,Y,Z) :- Z < Y, times(Y,Q,W), plus(W,Z,X).
mod(X,Y,X) :- X < Y.
mod(X,Y,Z) :- plus(X1,Y,X), mod(X1,Y,Z).
```

Example

```
ackermann(0,N,s(N)).
ackermann(s(M),0,Val) :- ackermann(M,s(0),Val).
ackermann(s(M),s(N),Val) :- ackermann(s(M),N,Val1),
    ackermann(M,Val1,Val).
```


Lists

Notation

• []	empty list	
• [H|T]	list with head H and tail T	
• [A]	[A|[]]	list with one element
• [A,B]	[A|[B|[]]]	list with two elements
• [A,B|T]	[A|[B|T]]	list with at least two elements

Example

```
is_list([]).  is_list([X|Xs]) :- is_list(Xs).
```

Notation

formal object	cons pair syntax	element syntax
.(a,[])	[a|[]]	[a]
.(a,.(b,[]))	[a|[b|[]]]	[a,b]

Example

```
member(X,[X|Xs]).
member(X,[Y|Xs]) :- member(X,Xs).    :- member(X,[a,b,a]).
```

Example

```
append(Xs,Ys,Zs) :-
    Xs = [],
    Zs = Ys.
append(Xs,Ys,Zs) :-
    Xs = [H|Ts],
    append(Ts,Ys,Us),
    Zs = [H|Us].
```

```
append([],Ys,Ys).
append([H|Ts],Ys,[H|Zs]) :-
    append(Ts,Ys,Zs).
```

Example

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

```
suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) :-
    suffix(Xs,Ys).
```


Example (non termination)

```
:-& infinite.

infinite :- infinite.
```

Example (again, but different)

```
:-& winfinite.

winfinite :- winfinite.
winfinite.
```

Example (non termination yet again)

```
:- hinfinite.
:-& hinfinite, false.

hinfinite.
hinfinite :- hinfinite.
```


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

equivalently

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


Example

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

- Ancestor doesn't occur in first goal (= recursive call)
- no influence on termination behaviour
- Descendant remains unchanged
- last goal has no effect

Example (specialised and generalised)

```
:-& ancestor_of_2(Ancestor, Descendant).

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

Strong vs Weak Normalisatin

Fact

suppose the solution set is infinite, then the query

```
:- Goal, false.
```

cannot terminate

Example (specialised and generalised)

```
:- child_of(X,X).
:- ancestor_of(Ancestor, Descendant), false.

ancestor_of(Ancestor, Descendant) :-
    child_of(Person, Ancestor),
    ancestor_of(Person, Descendant).
```


Terminationqueries

Definition

(non) termination queries:

- :/− Goal, false . — goal terminates
- :/−& Goal, false . — goal does not terminate
- :/−$ Goal, false . — goal terminates, but is expensive

Example (useful termination queries)

```
:- ancestor_of_2(Ancestor, Descendant).
:/-& ancestor_of_2(Ancestor, Descendant), false.
```