Logic Programming

Georg Moser

Department of Computer Science @ UIBK

Winter 2016

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).
```


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

Non-Monotonic Reasoning

Definition

an operator Ψ is called monotone if $A \subseteq B$ implies $\Psi(A) \subseteq \Psi(B)$

Fact

suppose Ψ acts on sets of formulas and interprets the consequence relation of a logic program P without negation, then Ψ is monotone

Definition

a monotone logic program is a logic program without negation (\+)

Monotonicity Criticism

Example (Minsky's Example)

```
on(a,b).
on(d,a).
on(d,c).
clear(Y) :-
    not_exists_x_on(Y).

not_exists_x_on(Y) :-
    on(_X,Y), !, fail.
not_exists_x_on(_Y).
```

Observations

- in this block-world example `:- clear(d)` holds
- but monotoncity doesn't; addition of the fact `on(e,d).` renders `:- clear(d)` false

Theory of Monotone Logic Programs

Definitions

- goal clause

$$:\!-\ B_1, \dots, B_n$$

consists of sequence $B_1, \dots, B_n$ of goals
- empty goal clause `:-` is denoted by $\square$
- resolvent of goal clause $:\!-\ B_1, \dots, B_i, \dots, B_m$ and rule $A\ :\!-\ A_1, \dots, A_n$ is goal clause

$$:\!-\ B_1\sigma, \dots, B_{i-1}\sigma, A_1\sigma, \dots, A_n\sigma, B_{i+1}\sigma \dots, B_m\sigma$$

provided B_i (selected goal) and A unify with most general unifier σ

NB: see week 2 for the most general unifier

Selective Linear Definite Clause Resolution

Definitions

- SLD-derivation of logic program P and goal clause G consists of
  1. maximal sequence $G_0, G_1, G_2, \dots$ of goal clauses
  2. sequence $C_0, C_1, C_2, \dots$ of variants of rules in P
  3. sequence $\sigma_0, \sigma_1, \sigma_2, \dots$ of substitutions

  such that
  - $G_0 = G$
  - G_{i+1} is resolvent of G_i and C_i with mgu σ_i
  - C_i has no variables in common with $G, C_0, \dots, C_{i-1}$
- SLD refutation is finite SLD derivation ending in $\square$
- computed answer substitution of SLD refutation of P and G with substitutions $\sigma_0, \sigma_1, \dots, \sigma_m$ is restriction of $\sigma_0\sigma_1 \cdots \sigma_m$ to variables in G

Example

```
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).
:- times(X,X,Y)
```

SLD-refutation

G_0: `:- times(X,X,Y)`
- C_0: $\texttt{times(s(X}_0\texttt{),Y}_0\texttt{,Z}_0\texttt{) :- times(X}_0\texttt{,Y}_0\texttt{,U}_0\texttt{), plus(U}_0\texttt{,Y}_0\texttt{,Z}_0\texttt{)}$
- σ_0: $X \mapsto s(X_0),\ Y_0 \mapsto s(X_0),\ Z_0 \mapsto Y$

G_1: $\texttt{:- times(X}_0\texttt{,s(X}_0\texttt{),U}_0\texttt{), plus(U}_0\texttt{,s(X}_0\texttt{),Y)}$
- C_1: $\texttt{times(0,X}_1\texttt{,0).}$
- σ_1: $X_0 \mapsto 0,\ X_1 \mapsto s(0),\ U_0 \mapsto 0$

G_2: `:- plus(0,s(0),Y)`
- C_2: $\texttt{plus(0,X}_2\texttt{,X}_2\texttt{).}$
- σ_2: $X_2 \mapsto s(0),\ Y \mapsto s(0)$

G_3: $\square$ computed answer substitution: $X \mapsto s(0),\ Y \mapsto s(0)$

Definition

- a selection function selects the next goal G in goal clause, where resolution is attempted
- Prolog's selection function proceeds left to right

Theorem

$\forall$ logic programs P and goal clause G
$\forall$ computed answer substitutions σ
$\forall$ selection functions $\mathcal{S}$
$\exists$ computed answer substitution σ' using $\mathcal{S}$
such that σ' is at least as general as σ (with respect to variables in G)

Search or SLD Trees

Definition

a search tree (aka SLD tree) of a goal G is a tree T such that

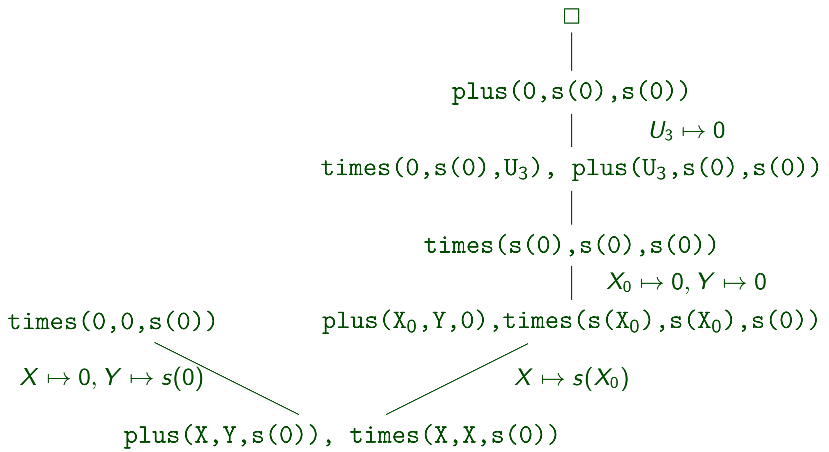
- the root of T is labelled with G
- the nodes of T are labelled with conjunctions of goals, where one goal is selected (wrt a selection function)
- for each clause, whose head unifies with the selected goal $\exists$ edge from node N
- edges are labelled with (partial) answer substitutions
- leaves are success nodes, if the empty goal (denoted by $\square$) has been reached or failure nodes otherwise

Remark

a search tree captures all possible SLD derivations wrt a given goal and selection function

Example (cont'd)

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



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

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]