Programming Paradigms

Summer Term 2017

12th Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(_),0,0).
mult(s(X),s(Y),s(Z)) :- mult(X,s(Y),U), add(Y,U,Z).
```

$$T_P(\emptyset) = \{\texttt{add(0,0,0)}, \texttt{add(0,s(0),s(0))}, \ldots\} \cup \{\texttt{mult(0,0,0)}, \texttt{mult(0,s(0),0)}, \ldots\} \cup \{\texttt{mult(s(0),0,0)}, \ldots\}$$

$$T_P(T_P(\emptyset)) = T_P(\emptyset) \cup \{\texttt{add(s(0),0,s(0))}, \texttt{add(s(0),s(0),s(s(0)))}, \ldots\} \cup \{\texttt{mult(s(0),s(0),s(0))}\}$$

$$T_P(T_P(T_P(\emptyset))) = T_P(T_P(\emptyset)) \cup \{\texttt{add(s(s(0)),0,s(s(0)))}, \ldots\} \cup \{\texttt{mult(s(0),s(s(0)),s(s(0)))}, \texttt{mult(s(s(0)),s(0),s(s(0)))}\}$$

$$T_P^4(\emptyset) = T_P^3(\emptyset) \cup \{\texttt{add(}s^3\texttt{(0),0,}s^3\texttt{(0))}, \texttt{add(}s^3\texttt{(0),s(0),}s^4\texttt{(0))}, \ldots\} \cup \{\texttt{mult(s(0),}s^3\texttt{(0),}s^3\texttt{(0))}, \texttt{mult(}s^2\texttt{(0),}s^2\texttt{(0),}s^4\texttt{(0))}, \texttt{mult(}s^3\texttt{(0),s(0),}s^3\texttt{(0))}\}$$

Applicability of the semantics based on Herbrand models

The declarative semantics:

- is only applicable to certain, “purely logical”, programs
- does not directly describe the behaviour for queries containing variables
- is mathematically simpler than the still to be introduced operational semantics
- can be related to that operational semantics appropriately
- is insensitive against changes to the order of, and within, facts and rules (!)

Programming Paradigms

Operational semantics of Prolog

Motivation: Observing some not so nice (not so "logical"?) effects

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

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

```
?- connection(frankfurt,maui).
true.

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

?- connection(maui,X).
false.
```

Motivation: Observing some not so nice (not so “logical”?) effects

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

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

```
?- connection(frankfurt,maui).
ERROR: Out of local stack
```

- Apparently, the implicit logical operations are not commutative.
- So underlying the program execution, there must be more than the purely logical reading.

Somewhat more subtle…

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```
?- sub(N,M,s(0)).
N = s(M) ;
false.
```

```
add(X,0,X).
add(X,s(Y),s(Z)) :- add(X,Y,Z).

sub(X,Y,Z) :- add(Z,Y,X).
```

```
?- sub(s(s(0)),s(0),N).
N = s(0) ;
false.

?- sub(N,M,s(0)).
N = s(0),
M = 0 ;
N = s(s(0)),
M = s(0) ;
```

So the choice/treatment of the order of arguments in definitions affects the quality of results.

…

… and (thus) sometimes less flexibility than desired

The nicely descriptive solution:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

works very well for several kinds of queries:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(s(0)))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

One says that **mult** supports the "call modes" **mult(+X,+Y,?Z)** and **mult(+X,?Y,+Z)**

But there are also "outliers":

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```

… but not **mult(?X,?Y,+Z)**.

otherwise infinite search

… and (thus) sometimes less flexibility than desired

Whereas with just addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

the analogous call mode seemed to work pretty well:

```
?- add(N,M,s(s(s(0)))).
N = 0,
M = s(s(s(0))) ;
N = s(0),
M = s(s(0)) ;
N = s(s(0)),
M = s(0) ;
N = s(s(s(0))),
M = 0 ;
false.
```

Indeed, **add** supports all call modes, even **add(?X,?Y,?Z)**.

1. So why the difference?

2. And what can one do to also let **mult** function this way?

Moreover, caution needed when using/positioning negative literals

And now it gets really "strange":

```
loves(vincent,mia).
loves(marsellus,mia).
loves(mia,vincent).

jealous(X,Y) :- loves(X,Z), loves(Y,Z), X \= Y.
```

small change

```
...

jealous(X,Y) :- X \= Y, loves(X,Z), loves(Y,Z).
```

```
?- jealous(marsellus,X).
false.

?- jealous(X,_).
false.

?- jealous(X,Y).
false.
```

Whereas before the small change, we got meaningful results for these queries!

To investigate all these phenomena, we have to consider the concrete execution mechanism of Prolog.

Ingredients for this discussion of the operational semantics, considered in what follows:

1. Unification
2. Resolution
3. Derivation trees

Programming Paradigms

Unification

Analogy to Haskell: Pattern matching

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

```
?- add(s(s(0)),s(0),s(s(s(0)))).
?- add(s(0),s(0),s(s(0))).
?- add(0,s(0),s(0)).
?- .
true.
```

But what about "output variables"?

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

?

```
?- add(s(s(0)),s(0),N).
```

Unification as “bidirectional pattern matching”

Equality “`=`” as binary Prolog predicate that accomplishes a lot:

- performing comparisons on ground terms (terms without variables), e.g.:
  - `s(0)=s(0)` ⇒ `true`
  - `s(0)=s(s(0))` ⇒ `false`
- accepting bindings of variables, e.g.:
  - `N=0` ⇒ `true`
  - `N=s(U)` ⇒ `true`
  - `s(0)=N` ⇒ `true`
  - `M=V` ⇒ `true`
- structurally matching and binding, e.g.:
  - `s(s(0))=s(V)` ⇒ `V=s(0)`
  - `s(U)=s(0)` ⇒ `U=0`
- “collecting”/combining bindings, e.g.:
  - `N=s(V), M=V` ⇒ `N=s(M)`

Equality of terms (1)

- Checking equality of ground terms:

```
europe = europe ?                                      yes
person(fritz,mueller) = person(fritz,mueller) ?        yes
person(fritz,mueller) = person(mueller,fritz) ?        no
5 = 2 ?                                                no
5 = 2 + 3 ?                                            no
2 + 3 = +(2, 3) ?                                      yes
```

⇒ Equality of terms means structural equality.

Terms are not “evaluated” before a comparison!

Equality of terms (2)

- Checking equality of terms with variables:

```
person(fritz, Lastname, datum(27, 11, 2007))
     = person(fritz, mueller, datum(27, MM, 2007)) ?
```

- For a variable, any term may be substituted:
  - in particular `mueller` for `Lastname` and `11` for `MM`.
  - After this substitution both terms are equal.

Equality of terms (3)

Which variables have to be substituted how, in order to make the terms equal?

```
date(1, 4, 1985) = date(1, 4, Year) ?

date(Day, Month, 1985) = date(1, 4, Year) ?

a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,g(H,i,K))) ?

X = Y + 1 ?

[[the, Y]|Z] = [[X, dog], [is, here]] ?
```

As a reminder, list syntax:

```
[1,2,a] = [1|[2,a]] = [1,2|[a]] = [1,2|.(a,[])] = .(1,.(2,.(a,[])))
```

And what about:

```
p(X) = p(q(X)) ?
```

"occurs check" (see later)

Equality of terms (4)

Some further (problematic) cases:

```
loves(vincent, X) = loves(X, mia) ?

loves(marcellus, mia) = loves(X, X) ?

a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,p(H,i,K))) ?

p(b,b) = p(X) ?

...
```

Unification, formally (1)

Substitution:

- Replacing variables by other variables or other kinds of terms (constants, structures, …)
- A function which uniquely maps each term to a new term, where the new term differs from the old term only by replacement of variables.
- Notation: *U* = {`Lastname / mueller, MM / 11`}
- The substitution U changes only the variables `Lastname` and `MM`, everything else stays unchanged!
- *U*(`person(fritz, Lastname, datum(27, 11 2007))`
  `== person(fritz, mueller, datum(27, 11, 2007))`

Unification, formally (2)

- Unifier:
  - substitution that makes two terms equal
  - e.g., application of the substitution U = { `Lastname/mueller, MM/11` } :

    U(`person(fritz,Lastname,date(27,11 2007))`)
    == U(`person(fritz,mueller,date(27,MM,2007))`)

- Most general unifier:
  - unifier that leaves as many as possible variables unchanged
  - Example: `date(DD,MM,2007)` and `date(D,11,Y)`
    - U_1 = { `DD/27, D/27, MM/11, Y/2007` } ✗
    - U_2 = { `DD/D, MM/11, Y/2007` } ✓

- Prolog always looks for a most general unifier.

Unification, formally (3) – Computing a most general unifier

Input: two terms T_1 and T_2 (in general possibly containing common variables)

Output: a most general unifier U for T_1 and T_2 in case T_1 and T_2 are unifiable, otherwise failure

Algorithm:

1. If T_1 and T_2 are the same constant or variable, then $U = \emptyset$
2. If T_1 is a variable that does not occur in T_2, then $U = \{T_1 / T_2\}$
3. If T_2 is a variable that does not occur in T_1, then $U = \{T_2 / T_1\}$

"occurs check" (applies to 2. and 3.)

Unification, formally (4) – Computing a most general unifier

Algorithm (cont.):

4. If $T_1 = f(T_{1,1},\ldots,T_{1,n})$ and $T_2 = f(T_{2,1},\ldots,T_{2,n})$ are structures with the same functor and the same number of components, then

   1. Find a most general unifier U_1 for $T_{1,1}$ and $T_{2,1}$
   2. Find a most general unifier U_2 for $U_1(T_{1,2})$ and $U_1(T_{2,2})$

   …

   n. Find a most general unifier U_n for

   $U_{n-1}(\ldots(U_1(T_{1,n})\ldots)$ and $U_{n-1}(\ldots(U_1(T_{2,n}))\ldots)$

   If all these unifiers exist, then

   $U = U_n \circ U_{n-1} \circ \ldots \circ U_1$ (function composition of the unifiers)

5. Otherwise: T_1 and T_2 are not unifiable.

Unification – Examples

```
date(1, 4, 1985) = date(1, 4, Year) ?
```

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for `1` and `1`

   $\Rightarrow$ same constants, thus $U_1 = \varnothing$

2. Find a most general unifier U_2 for U_1(`4`) and U_1(`4`)

   $\Rightarrow$ same constants, thus $U_2 = \varnothing$

3. Find a most general unifier U_3 for $U_2(U_1($`1985`$))$ and $U_2(U_1($`Year`$))$

   $\Rightarrow$ constant vs. variable, thus $U_3 = \{$`Year`/`1985`$\}$

A most general unifier overall is:

$U = U_3 \circ U_2 \circ U_1 = \{$`Year`/`1985`$\}$

Unification – Examples

`loves(marcellus, mia) = loves(X, X) ?`

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for `marcellus` and `X`

   $\Rightarrow$ constant vs. variable, thus U_1 = {`X` /`marcellus`}

2. Find a most general unifier U_2 for U_1(`mia`) = `mia` and U_1(`X`) = `marcellus`

   $\Rightarrow$ different constants, hence U_2 does not exist!

Consequently, also no unifier exists for the original terms!

Unification – Examples

$$\texttt{d(E,g(H,J)) = d(F,g(H,E))} \ ?$$

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for **E** and **F**

   $\Rightarrow$ different variables, thus $U_1 = \{\mathtt{E}/\mathtt{F}\}$

2. Find a most general unifier U_2 for $U_1(\mathtt{g(H,J)})$ and $U_1(\mathtt{g(H,E)})$

   $$\texttt{g(H,J) = g(H,F)} \ ?$$

   $\Rightarrow$ Structures with the same functor, same number of components, hence:

   - Find a most general unifier $U_{2,1}$ for **H** and **H**

     $\Rightarrow$ same variables, thus $U_{2,1} = \emptyset$

   - Find a most general unifier $U_{2,2}$ for $U_{2,1}(\mathtt{J})$ and $U_{2,1}(\mathtt{F})$

     $\Rightarrow$ different variables, thus $U_{2,2} = \{\mathtt{F}/\mathtt{J}\}$

   $U_2 = U_{2,2} \circ U_{2,1} = \{\mathtt{F}/\mathtt{J}\}$

A most general unifier overall is:

$$U = U_2 \circ U_1 = \{\mathtt{E}/\mathtt{J}\ ,\ \mathtt{F}/\mathtt{J}\}$$

Relevance of the "occurs check"

As a reminder:

2. If T_1 is a variable that does not occur in T_2, then $U = \{T_1 / T_2\}$

3. If T_2 is a variable that does not occur in T_1, then $U = \{T_2 / T_1\}$

"occurs check"

So, for example:

```
X = q(X)  ?
```

$\Rightarrow$ No unifier exists.

But in Prolog this check is actually not performed by default!

Relevance of the "occurs check"

Without "occurs check":

`p(X) = p(q(X))` ?

Structures with the same functor, same number of components, hence:

1. Find a most general unifier U_1 for `X` and `q(X)`

   ⇒ variable vs. term, thus U_1 = {`X`/`q(X)`}

U = U_1 = {`X`/`q(X)`} !

Although it actually is <u>not</u> true that U(`p(X)`) and U(`p(q(X))`) are equal!

Programming Paradigms

Resolution

Resolution in Prolog (1)

Resolution (proof principle) – without variables

One can reduce proving the query

`?- P, L, Q.` (let **L** be a variable free literal and **P** and **Q** be sequences of such)

to proving the query

`?- P,` L_1`,` L_2`, ... ,` L_n`, Q.`

provided that `L :-` L_1`,` L_2`, ...,` L_n`.` is a clause in the program (where $n \geq 0$).

- The choice of the literal **L** and the clause to use are in principle arbitrary.
- If $n = 0$, then the query becomes smaller by the resolution step.

Resolution in Prolog (2)

Resolution – with variables

One can reduce proving the query

`?- P, L, Q.` (let **L** be a literal and **P** and **Q** be sequences of literals)

to proving the query

$$\texttt{?- } U(\texttt{P}), U(\texttt{L}_1), U(\texttt{L}_2), \ldots, U(\texttt{L}_n), U(\texttt{Q}).$$

provided that:

- there is a program clause $\texttt{L}_0 \texttt{:- } \texttt{L}_1, \texttt{L}_2, \ldots, \texttt{L}_n.$ (where n ≥ 0), with – just in case – renamed variables (so that there is no overlap with those in `P, L, Q`),

- and U is a most general unifier for **L** and $\texttt{L}_0$.

Programming Paradigms

Derivation trees

Reminder: Motivation for considering operational semantics…

We wanted to understand why, for example, for

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

several kinds of queries/“call modes” work very well:

```
?- mult(s(s(0)),s(s(s(0))),N).
N = s(s(s(s(s(s(0)))))).

?- mult(s(s(0)),N,s(s(s(s(0))))).
N = s(s(0)) ;
false.
```

but others don’t:

```
?- mult(N,M,s(s(s(s(0))))).
N = s(0),
M = s(s(s(s(0)))) ;
N = s(s(0)),
M = s(s(0)) ;
abort
```

otherwise infinite search

Explicit enumeration of solutions

Let us start with a simple example just for addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

Exhaustive search: