Programming Paradigms

Summer Term 2017

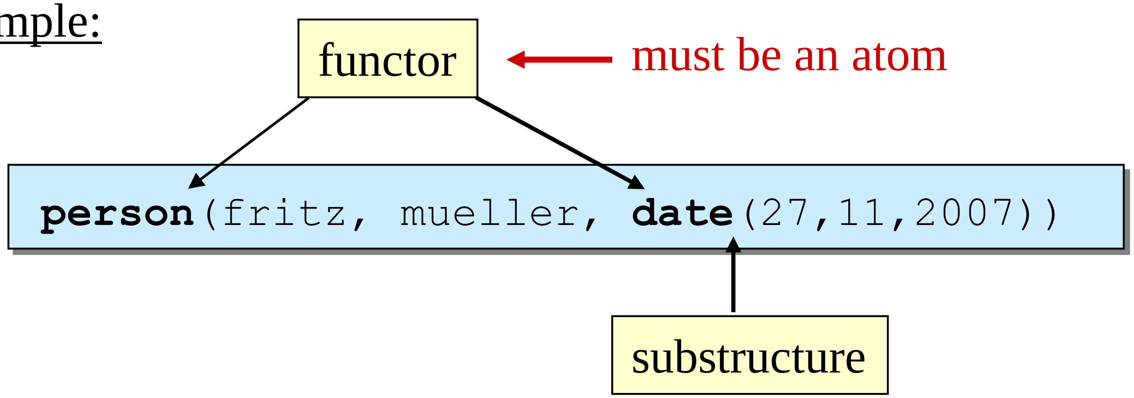
11^{th} Lecture

Prof. Janis Voigtländer
University of Duisburg-Essen

Structures in Prolog

- Structures represent objects that are made up of several other objects.

- Example:



functors: **person/3, date/3** (notation for arity)

- Through this, modelling of essentially "algebraic data types" – but not actually typed. So, **person(1,2,'a')** would also be a legal structure.

- Arbitrary nesting depth allowed – in principle infinite.

Language objects in Prolog

Predefined syntax for special structures:

- There is a predefined “list type” as recursive data structure:

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

- Character strings are represented as lists of ASCII-Codes:

```
"Prolog"   = [80, 114, 111, 108, 111, 103]
           = .(80,  . (114,  . (111,  . (108,  . (111,  . (103, [ ]))))))
```

Operators:

- Operators are functors made from symbols and written infix.
- Example: in arithmetic expressions
    - Mathematical functions are defined as operators.
    - `1 + 3 * 4` is to be read as this structure: `+(1,*(3,4))`

Language objects in Prolog

Collective notion "terms":

- Terms are constants, variables or structures:

```
fritz
27
MM
[europe, asia, africa | Rest]
person(fritz, Lastname, date(27, MM, 2007))
```

- A ground term is a term that does not contain variables:

```
person(fritz, mueller, date(27, 11, 2007))
```

Simple example for working with data structures

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

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

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

- Recall, in Haskell:

```
data Nat = Zero | Succ Nat

add :: Nat → Nat → Nat
add Zero      x = x
add (Succ x)  y = Succ (add x y)
```

Systematic connection/derivation?

- Essential difference Haskell/Prolog:

Functions	vs.	Predicates/Relations
f x y = z	"corresponds to"	`p(X,Y,Z).`

- First a somewhat naïve attempt to exploit this correspondence:

```
add Zero      x = x
```

↓

add(Zero, x, x)

↓

```
add(0,X,X).
```

```
add (Succ x)  y = Succ (add x y)
```

↓

add(Succ x, y, Succ (add x y))

↓

???

Systematic connection/derivation?

- Essential difference Haskell/Prolog:

Functions vs. Predicates/Relations

f x y = z "corresponds to" `p(X,Y,Z).`

- Systematically avoiding nested calls:

add (Succ x) y = Succ (add x y)

↓

add (Succ x) y = Succ z where z = add x y

↓

add(Succ x, y, Succ z) if add(x, y, z)

↓

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

On the flexibility of Prolog predicates

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

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

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

?- add(N,M,O).
```

???

On the flexibility of Prolog predicates

```
add(0,X,X).
add(s(X),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(M) ;
false.
```

Another example

Computing the length of a list in Haskell:

```
length []       =  0
length (x:xs)   =  length xs + 1
```

Computing the length of a list in Prolog:

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

```
?- length([1,2,a],Res).
   Res = 3.

?- length(List,3).
   List = [_G331, _G334, _G337]
```

list with 3 arbitrary (variable) elements

Arithmetics vs. symbolic operator terms

Careful: If instead of:

```
length([],0).
length([X|Xs],N) :- length(Xs,M), N is M+1.
```

we use:

```
length([],0).
length([X|Xs],M+1) :- length(Xs,M).
```

then:

```
?- length([1,2,a],Res).
   Res = 0+1+1+1.

?- length(List,3).
   false.

?- length(List,0+1+1+1).
   List = [_G331, _G334, _G337].
```

An example corresponding to several nested calls

partition :: Int → [Int] → ([Int], [Int])
…

quicksort [ ] = [ ]
quicksort (h : t) = quicksort l_1 ++ h : quicksort l_2
  where (l_1, l_2) = partition h t

↓

quicksort [ ] = [ ]
quicksort (h : t) = ls ++ h : quicksort l_2
  where (l_1, l_2) = partition h t
    ls = quicksort l_1

↓

quicksort [ ] = [ ]
quicksort (h : t) = ls ++ h : lg
  where (l_1, l_2) = partition h t
    ls = quicksort l_1
    lg = quicksort l_2

→

quicksort [ ] = [ ]
quicksort (h : t) = list
  where (l_1, l_2) = partition h t
    ls = quicksort l_1
    lg = quicksort l_2
    list = ls ++ h : lg

↑

```
quicksort([], []).
quicksort([H|T], List) :-
   partition(H, T, L1, L2),
   quicksort(L1, LS),
   quicksort(L2, LG),
   append(LS, [H|LG], List).
```

Programming Paradigms

Declarative semantics of Prolog

Declarative semantics of Prolog

What is the "mathematical" meaning/semantics of a Prolog program?

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

Logical interpretation:

$$(\forall\ X.\ add(0,X,X))$$
$$\wedge\ (\forall\ X, Y, Z.\ add(X,Y,Z)\ \Rightarrow\ add(s(X),Y,s(Z)))$$

To give meaning to such formulas, the study of logics uses models:

- starting from a set of mathematical objects
- interpretation of constants (like "0") as elements of the above set, and of functors (like "s(…)") as functions thereover
- interpretation of predicates (like "add(…)") as relations between objects
- assignment of truth values to formulas according to certain rules
- consideration only of interpretations that make all given formulas true

Semantics of a program would be given by all statements/relationships that hold in all models for the program.

Some example models

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

$$(\forall\ X.\ add(0,X,X))$$
$$\wedge\ (\forall\ X, Y, Z.\ add(X,Y,Z)\ \Rightarrow add(s(X),Y,s(Z)))$$

Model 1:
objects: natural numbers
interpretation of 0 as 0
interpretation of s(…) as $s(n) = n + 1$
interpretation of add(…) as: add(n,m,k) if and only if $n + m = k$

Model 2:
objects: $\{*\}$
interpretation of 0 as $*$
interpretation of s(…) as $s(*) = *$
interpretation of add(…) as: add($*$,$*$,$*$) is true

Model 3:
objects: non-positive integers
interpretation of 0 as 0
interpretation of s(…) as $s(n) = n - 1$
interpretation of add(…) as: add(n,m,k) if and only if $n + m = k$

Herbrand models

Important: There is always a kind of "universal model".

Idea: Interpretation as simple as possible, namely purely syntactic.
Neither functors nor predicates really "do" anything.

the Herbrand universe

So:
- set of objects = all ground terms (over implicitly given signature)
- interpretation of functors = syntactical application on terms
- interpretation of predicates = assigning some set of applications of predicate symbols on ground terms

a Herbrand interpretation

Example:

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

Signature: **0** (of arity 0), **s** (of arity 1)
Herbrand universe: {**0**, **s(0)**, **s(s(0))**, **s(s(s(0)))**, …} (without predicate symbols!)
the Herbrand base: {**add(0,0,0)**, **add(0,0,s(0))**, **add(0,s(0),0)**, …}
(all applications of predicate symbols on terms from Herbrand universe)

Herbrand models

A Herbrand interpretation is some subset of the Herbrand base.

Example:

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

Herbrand interpretation 1: {`add(0,0,0)`, `add(0,0,s(0))`, `add(0,s(0),0)`, …}

Herbrand interpretation 2: $\emptyset$

Herbrand interpretation 3: {`add(0,0,0)`, `add(0,s(0),s(0))`,
`add(s(0),0,s(0))`, `add(s(0),s(0),s(s(0)))`, …}

Our aim is a Herbrand interpretation that makes true all formulas given by the program, but does not unnecessarily make anything else additionally true.

Herbrand models

A Herbrand interpretation is a model for a program if for every complete instantiation (i.e., no variables left)

$$\mathtt{L_0 \text{ :- } L_1,\ L_2,\ \ldots,\ L_n}$$

of each clause it holds: if $\mathtt{L_1,\ L_2,\ \ldots,\ L_n}$ is in the interpretation, then so is $\mathtt{L_0}$.

Example:

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

- The Herbrand base is (always) a model.
- The Herbrand interpretation ∅ = {} is (here) no model.
- The interpretation {**add(0,0,0)**, **add(0,s(0),s(0))**, **add(s(0),0,s(0))**, **add(s(0),s(0),s(s(0)))**, …} is here a model.

Smallest Herbrand model

The declarative meaning of a Prolog program is its smallest Herbrand interpretation that is a model!

For the example:

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

{`add(0,0,0)`, `add(0,s(0),s(0))`, `add(s(0),0,s(0))`, `add(s(0),s(0),s(s(0)))`, …}

Generally:

Is there always such a smallest model?

Yes, since models for programs consisting of so-called Horn clauses (exactly the kind of clauses in Prolog without negation) are closed under intersection!

Smallest Herbrand model

Can one actually compute, in a constructive fashion, the smallest Herbrand model?

Yes, using the "immediate consequence operator": T_P

Definition: T_P takes an interpretation I and produces all ground literals (elements of the Herbrand base) L_0 for which $L_1, L_2, ..., L_n$ exist in I such that L_0 :- $L_1, L_2, ..., L_n$ is a complete instantiation of any of the given program clauses.

Obviously: A Herbrand interpretation I is a model if and only if $T_P(I)$ is a subset of I.

Moreover: The smallest Herbrand model is obtained as fixpoint/limit of the sequence

$$\emptyset\ ,\ T_P(\emptyset)\ ,\ T_P(T_P(\emptyset))\ ,\ T_P(T_P(T_P(\emptyset)))\ ,\ \ldots$$

Smallest Herbrand model

On the example:

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

$T_P(\emptyset)$ = {`add(0,0,0)`, `add(0,s(0),s(0))`, `add(0,s(s(0)),s(s(0)))`, …}

$T_P(T_P(\emptyset))$ = $T_P(\emptyset) \cup$ {`add(s(0),0,s(0))`, `add(s(0),s(0),s(s(0)))`,
`add(s(0),s(s(0)),s(s(s(0))))`, …}

$T_P(T_P(T_P(\emptyset)))$ = $T_P(T_P(\emptyset)) \cup$ {`add(s(s(0)),0,s(s(0)))`,
`add(s(s(0)),s(0),s(s(s(0))))`,
`add(s(s(0)),s(s(0)),s(s(s(s(0)))))`, …}

…

Applicability of the semantics based on Herbrand models

For which kind of Prolog programs can one work with the T_P -semantics?

- no arithmetics, no **`is`**
- no **`\=`**, no **`not`**
- generally, none of the “non-logical” features (not yet introduced in the lecture)

But for example programs like:

```
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).
```

$T_P(\emptyset)$ = {**`add(0,0,0)`**, **`add(0,s(0),s(0))`**, …} ∪ {**`mult(0,0,0)`**, **`mult(0,s(0),0)`**, …} ∪ {**`mult(s(0),0,0)`**, …}

$T_P(T_P(\emptyset))$ = $T_P(\emptyset)$ ∪ {**`add(s(0),0,s(0))`**, **`add(s(0),s(0),s(s(0)))`**, …} ∪ {**`mult(s(0),s(0),s(0))`**}