

# Principles of Programming Languages

## Logic Programming

---

We introduced a restricted form of Logic Programming called *Relational Logic Programming* in the previous lecture. We noted that RLP is decidable (and hence not Turing-complete). RLP is still expressive enough to describe full relational algebra operations extended with recursive operations (such as transitive closure).

We now move on to full Logic Programming - and present a programming language which is Turing-complete. The increment from RLP to LP is remarkably small in terms of language design: we simply introduce **recursive terms**, using a new compound syntactic construct we call a **functor**. Functors enable LP to describe recursive data structures.

As a consequence of introducing recursive terms:

1. A more complex unification operation must be defined - we must introduce the *occurs-check* restriction to avoid loops.

- 
2. The language becomes only partially decidable. That is, while the answer to a query in relational logic programming can always be decided to be a success or a failure, logic programming is partially decidable, like all other general purpose programming languages.

We present the updated syntax (concrete and abstract) to support recursive terms and the adjustments required in the operational semantics to extend RLP into LP. We then describe how to model abstract data structures in Logic Programming, and specifically how to manage lists.

The only difference between the syntax of Logic Programming and the syntax of Relational Logic Programming is the addition of a new kind of a constant symbol: Functor (function symbol). It enriches the set of terms so that they can describe structured data.

The syntax of terms is now inductive:

1. *Base case*: Individual constant symbols and variables are terms (these are the only cases that exist in RLP).
2. *Inductive step*: For terms  $t_1, \dots, t_n$  and a functor  $f$ ,  $f(t_1, \dots, t_n)$  is a term.

- `'[|]'`(`a`, `[]`) describes the list `[a]`. `[]` is an individual constant, standing for the empty list. The `'[|]'` functor has a syntactic sugar notation as an infix operator `|`: `'[|]'`(`a`, `[]`) is written `[a | []]`.

## Examples of Composite Terms

- `'[|]'`(b, `'[|]'`(a, [])) is the list [b, a], or [b | [a | []]]. The syntax [b, a] uses the printed form of lists in Prolog.
- `'[|]'`(`'[|]'`(a, []), `'[|]'`(b, `'[|]'`(a, []))) – the list [[a], b, a], or [[a | []] | [b | [a | []]]] .

## Examples of Composite Terms

- `time(monday, 12, 14)`
- `street(alon, 32)`
- `tree(Element, Left, Right)` – a binary tree, with `Element` as the root, `Left` and `Right` as its sub-trees.
- `tree(5, tree(8, void, void), tree(9, void, tree(3, void, void)))`

## Atomic Formulas with Composite Terms

The arguments in atomic formulas can now be composite terms. For example:

- `father(abraham, isaac)`
- `p(f(f(f(g(a, g(b, c))))))`
- `ancestor(mary,  
sister_of(friend_of(john)))`
- `append([a, b], [c, d], [a, b, c,  
d])`

Every functor has an arity, that is, it specifies the number of arguments the functor expects.

In the examples above:

- The arity of `'[|]'` is 2
- The arity of `sister_of` is 1
- The arity of `time` is 3
- The arity of `street` is 2

## Nested Functors vs. Predicates

Functors can be nested: Terms might have unbound depth:  $f(f(f(g(a, g(b, c))))))$ . The number of different atomic formulas that can be constructed from a given finite set of predicate, functor and individual constant symbols is unbounded - in contrast to Relational Logic Programming. (Observe that the BNF of RLP presented in the previous Section is not recursive).

## Nested Functors vs. Predicates

In contrast to terms, predicate symbols cannot be nested:

- $p(f(f(f(g(a, g(b, c))))))$  –  $p$  is a predicate symbol, while  $f$  and  $g$  are functors.
- `ancestor(mary, sister_of(friend_of(john)))` – `ancestor` is a predicate symbol, and `sister_of` and `friend_of` are functors.

## Nested Functors vs. Predicates

- `course(ppl, time(tuesday, 14, 16), location(zoom))` – `course` is a predicate symbol, and `time` and `location` are functors.
- `address(street(alon, 32), shikun_M, tel_aviv, israel)` – `address` is a predicate symbol, and `street` is a functor.

The syntax of terms and of atomic formulas is identical. They differ in the position (context) in statements:

- Terms are arguments to either terms and to predicates.
- Atomic formulas appear inside rules and facts.

To support composite terms, we add the following concrete syntax rules:

```
<term> ::= <constant> | <variable> | <composite-term>  
<composite-term> ::= <functor> '(' (<term>',' )* <term>')'  
<functor> ::= <constant>
```

The corresponding abstract syntax for terms is adapted is:

```
<term>: <constant> | <variable> | <composite-term>  
<composite-term>: {functor: <constant>, args: <term>[]};
```

This last rule is the first recursive data type in the AST of Logic Programming we have met.

The **answer-query** abstract interpreter, presented in the previous lecture on Relational Logic Programming, applies to Logic Programming as well. The only difference is that the unification algorithm has to be extended to handle composite terms.

The presence of function symbols complicates the unification step in the abstract interpreter. Recall that the rule selection procedure tries to unify a query goal (an atomic formula) with the head of the selected rule (an atomic formula).

The unification operation, if successful, produces a substitution (most general unifier) for the variables in the atomic formulas.

The notion of substitution is modified to support composite terms by adding the **occurs-check** condition:

## Definition: Substitution with Composite Terms

A substitution  $s$  is a finite mapping from variables to terms, such that  $s(X)$  does not include  $X$ . All the rest of the substitution and unification terminology stays unchanged.

Observe the difference with the definition we had for RLP:

# Unification of Composite Terms

## Definition: Substitution with Composite Terms

A substitution  $s$  in logic programming involves logic variables as variables and logic terms as values, such that  $s(X) \neq X$ .

- For atomic terms, the condition is the simple  $s(X) \neq X$
- For composite terms, the condition is:  $s(X)$  does not include  $X$

Testing the condition “ $s(X)$  does not include  $X$ ” is an expensive computational operation (it is an unbounded operation because composite terms can have unbounded depth).

Occurs-check makes unification expensive.

## Examples of Unification with Composite Terms

```
unify(member(X, tree(X, Left, Right)),  
      member(Y, tree(9, void, tree(3, void, void))))  
-> { X = 9, Y = 9, Left = void, Right = tree(3, void, void)}
```

```
unify(member(X, tree(9, void, tree(E1, L1, R1)),  
      member(Y, tree(Y, Z, tree(3, void, void))))  
-> { X = 9, Y = 9, Z = void, E1 = 3, L1 = void, R1 = void }
```

## Examples of Unification with Composite Terms

```
unify(t(X, f(a), X),  
      t(g(U), U, W))  
-> { X = g(f(a)), U = f(a), W = g(f(a)) }
```

```
unify(t(X, f(X), X),  
      t(g(U), U, W))  
-> FAIL (occurs-check): X = g(U), U = f(X), X = g(f(X))
```

## Examples of Unification with Composite Terms

```
unify(append([1, 2, 3], [3, 4], List),  
      append([X | Xs], Ys, [X | Zs]))  
-> { X = 1, Xs = [2, 3], Ys = [3, 4], List = [1 | Zs] }
```

```
unify(append([1, 2, 3], [3, 4], [3, 3, 4]),  
      append([X | Xs], Ys, [Xs | Zs]))  
-> FAIL: Xs = [2, 3], Xs = 3
```

Logic Programming has the expressive power of Turing machines. That is, every computable program can be written in Logic Programming. In particular, every Scheme program can be written in Prolog, and vice versa.

Logic Programming is only partially decidable - unlike Relational Logic Programming. That is, the problem “Is  $Q$  provable from  $P$ ”, denoted  $P \vdash Q$ , is partially decidable. The finiteness argument of Relational Logic Programming does not apply here since in the presence of recursive terms, the number of different atomic formulas is unbounded (since terms can be nested up to unbounded depth).

Therefore, terminating proofs can have an unbounded length - even with a finite vocabulary of functors, predicates and constants.

Composite terms allow us to define abstract data types in LP. This requires some change of habit though - because we do not compute terms with functions that construct new complex values. Instead, when we program in LP, we define **relations** among values. When we check whether two values stand in relation, we instantiate some logical variables to make the predicate hold. When this happens, we end up constructing complex values.

Let us consider this strategy with a Tree data structure.

The predicate `binary_tree` corresponds to the membership predicate of values to the `binary_tree` datatype. It holds only for values which belong to the tree datatype. Values of this type are composite terms with the functor `tree/3`. This functor has no primitive (pre-defined) semantic. We define its semantic by using it in specific predicates.

```
% Signature: binary_tree(T)/1
% Purpose: T is a binary tree.
binary_tree(void).
binary_tree(tree(Element, Left, Right)) :-
    binary_tree(Left),
    binary_tree(Right).
```

We define recursive procedures over such composite values as relations. The relation `tree_member(X, T)` holds when `X` is bound to a value that occurs as one of the nodes in `T`.

```
% Signature: tree_member(X, T)/2
% Purpose: X is a member of T.
tree_member(X, tree(X, _, _)).
tree_member(X, tree(_, Left, _)) :-
    tree_member(X, Left).
tree_member(X, tree(_, _, Right)) :-
    tree_member(X, Right).
```

## Example Queries with Trees

```
?- tree_member(g(X),  
               tree(g(a),  
                   tree(g(b), void, void)  
                   tree(f(a), void, void))).
```

```
X = a;
```

```
X = b;
```

```
false
```

## Example Queries with Trees

```
?- tree_member(a, Tree).  
Tree = tree(a, _11426, _11428) ;  
Tree = tree(_12086, tree(a, _12096, _12098), _12090) ;  
...
```

Observe that in the first query above, the parameter **X** contains a variable, while the parameter **Tree** is completely bound to constants (no variables). In contrast, in the second query, **X** is bound to a constant (**a**) while **Tree** is bound to a variable.

The same procedure can be invoked in different ways - which are called **modes** - depending on which parameter is bound to constants and which is bound to variables.

In this example, the procedure `tree_member`, when it is invoked in the second mode with a free variable `Tree` is used as a **generator** of values. (In this example, the procedure is the generator of all possible trees which contain `a`.)

In LP, we did not introduce number terms - only symbols. Prolog extends LP with arithmetic and numeric terms. There is, however, a theoretical method to represent natural numbers using only symbols - called Church Numeral Encodings.

In Church encoding, natural numbers are represented by terms constructed from the symbol  $\mathbf{0}$  and the functor  $\mathbf{s}/\mathbf{1}$ :

- $\mathbf{0}$  denotes zero
- $\mathbf{s}(\mathbf{0})$  denotes one
- $\mathbf{s}(\mathbf{s}(\mathbf{s}(\dots\mathbf{s}(\mathbf{0})\dots)))$   $n$  times denotes  $n$
- In general, if  $\mathbf{N}$  is the Church encoding of  $n$ , then  $\mathbf{s}(\mathbf{N})$  is the Church encoding of  $n + 1$ .

### Membership predicate:

```
% Signature: natural_number(N)/1  
% Purpose: N is a natural number.  
natural_number(0).  
natural_number(s(X)) :- natural_number(X).
```

### Arithmetic Operations on Church Numerals:

```
% Signature: plus(X, Y, Z)/3  
% Purpose: Z is the sum of X and Y.  
plus(X, 0, X) :- natural_number(X).  
plus(X, s(Y), s(Z)) :- plus(X, Y, Z).
```

Example queries:

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

By changing the mode of the query, we  
compute subtraction:

```
?- plus(X, s(0), s(s(0))).  
X = s(0).
```

By using a more abstract mode, we obtain a generator of pairs of numbers:

```
?- plus(X, Y, s(s(0))).  
X = 0,          Y = s(s(0));  
X = s(0),       Y = s(0);  
X = s(s(0)),    Y = 0.
```

Natural number binary relation - Less than or equal:

```
% Signature: le(X,Y)/2  
% Purpose: X is less or equal Y.  
le(0, X) :- natural_number(X).  
le(s(X), s(Z)) :- le(X, Z).
```

Natural numbers multiplication:

```
% Signature: times(X,Y,Z)/2
% Purpose: Z = X*Y
times(0, X, 0) :- natural_number(X).
times(s(X), Y, Z) :-
    times(X, Y, XY),
    plus(XY, Y, Z).
```

## Syntax for Lists:

Lists are defined inductively - using a special infix operator (`|`) and a special notation `[a, b, c]` as syntactic sugar for nested '`[]`'/2 terms:

- `[]` is the empty list.
- `[Head | Tail]` is syntactic sugar for the composite term '`[]`'(`Head`, `Tail`), where `Tail` is a list term.

Simple syntax for bounded length lists:

- $[a \mid []] = [a]$
- $[a \mid [b \mid []]] = [a, b]$

Membership predicate:

```
list([]).  
list([_ | Xs]) :- list(Xs).
```

## List Membership:

```
% Signature: member(X, List)/2  
% Purpose: X is a member of List.  
member(X, [X|_]).  
member(X, [_|Ys]) :- member(X, Ys).
```

```
?- member(a, [b, c, a, d]). % checks membership
?- member(X, [b, c, a, d]). % takes an element
                           % from a list
?- member(b, Z). % generates a list containing b
```

## List Concatenation:

% Signature: `append(List1, List2, List3)/3`

% Purpose: `List3` is the concatenation of

% `List1` and `List2`.

`append([], Xs, Xs).`

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

% concatenation of two lists

```
?- append([a,b], [c], X).
```

% finds a difference between lists

```
?- append(Xs, [a,d], [b,c,a,d]).
```

% divides a list into two lists

```
?- append(Xs, Ys, [a,b,c,d]).
```

## List Selection using append:

% (a) List prefix and suffix:

```
prefix(Xs, Ys) :- append(Xs, _, Ys).  
suffix(Xs, Ys) :- append(_, Xs, Ys).
```

% (b) Redefine member:

```
member(X, Ys) :- append(_, [X | _], Ys).
```

## List Selection using append:

% (c) Adjacent list elements:

```
adjacent(X, Y, Zs) :- append(_, [X, Y | _], Zs).
```

% (d) Last element of a list:

```
last(X, Ys) :- append(_, [X], Ys).
```

## List Reverse:

```
% (a) A recursive version:  
% Signature: reverse(List1, List2)/2  
% Purpose: List2 is the reverse of List1.  
reverse([], []).  
reverse([H | T], R) :-  
    reverse(T, S),  
    append(S, [H], R).
```

```
?- reverse([a, b, c, d], R).  
R = [d, c, b, a]
```

But, what about the other mode of this predicate:

```
?- reverse(R, [a, b, c, d]).
```

Starting to build the proof tree, we see that the second query is:

```
reverse(T1, S1), append(S1, [H1], [a, b, c, d]).
```

This query fails on the first rule, and needs the second. The second rule is applied four times, until four elements are unified with the four elements of the input list.

We can try reversing the rule body:

```
reverse([H | T], R) :-  
    append(S, [H], R),  
    reverse(T, S).
```

The new version gives a good performance on the last direction, but poor performance on the former direction.

Conclusion: Rule body ordering impacts the performance in various directions.

Typical error: Wrong “assembly” of resulting lists:

```
wrong_reverse([H | T], R):-  
    wrong_reverse(T, S),  
    append(S, H, R).
```

```
% (b) an iterative version:
% Signature: reverse(List1, List2)/2
% Purpose: List2 is the reverse of List1.
%           This version uses an additional
%           reverse helper procedure, that uses
%           an accumulator.
reverse(Xs, Ys) :-
    reverse_help(Xs, [], Ys).
reverse_help([X | Xs], Acc, Ys) :-
    reverse_help(Xs, [X | Acc], Ys).
reverse_help([], Ys, Ys).
```

The length of the single success path is linear in the list length, while in the former version it is quadratic.

The `reverse_help` procedure is a helper procedure that should not reside in the global name space. Unfortunately, Logic Programming does not support nesting of name spaces (like Scheme with `letrec`). All names reside in the global space.