Logic Programming

Rodica Potolea
Camelia Lemnaru

Lecture #5

Cluj-Napoca

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Agenda

- Side effects
- Trees
  - representation
  - basic operations
- Tree traversal
  - pre, in, post orders
- Tree transformation into an ordered list
  - forward
  - backward
  - can we do better?

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Side effects

- Allow for **persistent structure**
- Allow for **wider scope visibility** (modelling global variables; created in some predicate and visible elsewhere, without passing arguments)
- Allow for **metaprogramming**
- By using them, we exceed the pure Logic Programming (*operations NOT undone on backtracking*)

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Side effects – contd.

- Built in predicates
- *assert/retract* for adding/extracting ONE clause of the predicate with the name provided as argument
- The argument (for both assert and retract) is a variable representing a dynamic predicate
- `assert(X)/retract(X)`
- The variable (X) must be ALREADY instantiated to the predicate intended to be updated

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Side effects – contd.

- assert comes into 2 flavors:
  - asserta – adds in the beginning (on top) of the existing clauses for the predicate provided as argument
  - assertz – adds at the end (at bottom) of the existing clauses for the predicate provided as argument
  - It NEVER fails
  - on backtracking, the item (clause) is NOT removed. However, assert does not re-execute on backtracking (NO new clause is added).

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Side effects – contd.

- retract:
  - removes the first item (clause of the predicate provided as argument) with whom it matches successfully
  - if no match (the matching fails), retract fails. So, as opposed to assert, it could fail
  - on backtracking, the item (clause removed) is NOT added back. On the contrary, another item is removed in case another successful match is possible, *or* retract fails

assert

- Assume we have some (3 in the ex.) clauses for a predicate p() (arguments do not matter here).
- Here on the right, the clauses have numbers added to represent their index

p():-…	p_1():-…
p():-…	p_2():-…
p():-…	p_3():-…

asserta/z

- As a result of the execution of asserta(p()), the predicate becomes:

p_{new}():-…
p_1():-…
p_2():-…
p_3():-…

- As a result of the execution of assertz(p()), the predicate becomes:

p_1():-…
p_2():-…
p_3():-…
p_{new}():-…

retract

- Assume the same scenario:

$p_1()$:-…

$p_2()$:-…

$p_3()$:-…

- The execution of retract(p()) would:
  - Start scanning the list of clauses of the p
  - Stop at the first clause where arguments (not mentioned here) match
  - Remove the given clause and report success
  - If no successful matching, the retract fails

retract – success on the first clause

- Assume the same scenario:

$p_1()$:-…

$p_2()$:-…

$p_3()$:-…

- The execution of retract(p()) succeeds on matching with the **first** clause of p would lead to:

$p_2()$:-…

$p_3()$:-…

retract – success on the second clause

- Assume the same scenario:

$p_1()$:-…

$p_2()$:-…

$p_3()$:-…

- The execution of retract(p()) succeeds on matching with the **second** clause of p would lead to:

$p_1()$:-…

$p_3()$:-…

retract – success on the third clause

- Assume the same scenario:

$p_1()$:-…

$p_2()$:-…

$p_3()$:-…

- The execution of retract(p()) succeeds on matching with the **third** clause of p would lead to:

$p_1()$:-…

$p_2()$:-…

retract – fails

- Assume the same scenario:

$p_1()$:-…

$p_2()$:-…

$p_3()$:-…

- The execution of retract(p()) **does NOT succeed** trying to match p to ANY of the 3 clauses, the result would be:

$p_1()$:-…

$p_2()$:-…

$p_3()$:-…

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

retract – on backtracking

- After the execution of retract, on backtracking, NO undo is performed (the removal stays). That clause is NEVER restore, unless an explicit request (assert) is made
- Moreover, on the attempt to REsatisfy the goal (due to backtracking), Prolog continues FROM THE PLACE it previously removed and tries to remove ANOTHER clause, and:
  - it does so if the match succeeds,
  - or else fails and the execution is resumed (backtracked) to the predicate on the left of retract

retract – on backtracking - contd

- Assume the same scenario where we have 5 clauses:

$p_1()$:-…
$p_2()$:-…
$p_3()$:-…
$p_4()$:-…
$p_5()$:-…

- Assume the execution of retract(p()) succeeds on matching with the **second** clause of p, after execution the predicate contains:

$p_1()$:-…
$p_3()$:-…
$p_4()$:-…
$p_5()$:-…

retract – on backtracking - contd

- If now we backtrack again, and $\mathbf{p_3}$ is matched, the predicate looks:

$p_1()$:-…
$p_4()$:-…
$p_5()$:-…

- If we now backtrack again, and $\mathbf{p_5}$ is matched the predicate looks:

$p_1()$:-…
$p_4()$:-…

- A new backtrack now would fail, as we reached the end of the list of clauses.

retract – on backtracking - contd

- Assume the same scenario where we have 5 clauses:

$p_1()$:-…
$p_2()$:-…
$p_3()$:-…
$p_4()$:-…
$p_5()$:-…

- Assume the execution of retract(p()) cannot unify with **ANY** of the 5 clauses, the execution **fails** and p remains unchanged:

$p_1()$:-…
$p_2()$:-…
$p_3()$:-…
$p_4()$:-…
$p_5()$:-…

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

assert and retract

- Combining assert and retract
  - asserta + retract => LIFO
  - assertz + retract => FIFO

Remember:

!!! assert – ALWAYS succeeds, NEVER backtracks

!!! retract – may fail, backtracks !!!

Trees

- Structure not defined; it is (used) as we define them
- Traditional (in use in literature) definition is either prefix or infix notation, with a letter as name for the structure: n or t.
- Therefore, the tree

- Is represented in one or the forms:

prefix not: tree(n(7,n(2,nil,nil),n(9,nil,nil))).

infix not: tree(n(n(nil,2,nil),7,n(nil,9,nil))).

- Existing trees are represented as facts (predicate tree) in the program

Trees –operations

- Same as in other languages
- Basic operations:
  - Traversal
  - Search
  - Insert
  - Delete

Trees – traversal

- 3 types: pre/in/post order

```
inorder(nil).
inorder(n(Left,Key,Right)):-
        inorder(Left),
        do_something(Key),
        inorder(Right).
```

- The predicate has not outcome/output unless do_something is doing side effects. Why?
  - No arguments!
- The order of clauses doesn't matter. Why?
  - Due to indexation on the first argument
- The predicate do_something does the actual job
- It runs O(n)
  - in case do_something is having constant time (O(1))
- Otherwise the time is calculated with the Master Theorem.
- pre/post orders are similar, just that do_something goes first/last.

Tree transformation into an ordered list

- Given a Binary Search Tree
  - BST; what is it? Why do we represent trees as search trees?
  - Benefits? Limitations?
  - Means of overcoming limitation?
- Generate the ordered list of the keys in the tree
- Approaches: divide et impera
  - divide – already done (the tree structure is a partition already)
  - impera – combine the results obtained on the subsets of the partition
  - Depending on how efficient we combine them we get the efficiency of the solution
- Approaches considered:
  - Forward and backward recursion
  - Pre/in/post order (which one is available in the context)

Tree 2 list forward

- By the approach, besides the output argument, we need a supplementary argument, some partial result to accumulate the results "so far"
- The arguments needed are: input tree, output list, accumulator (list type)
- Order of arguments: input has to come first to benefit from the first argument indexation
- The other arguments are in any order

Tree 2 list forward – code

- By the approach, besides the output argument, we need a supplementary argument, some partial result to accumulate the results "so far"
- The arguments needed are: input tree, output list, accumulator (list type)
- Order of arguments: input has to come first to benefit from the first argument indexation
- The other arguments are in any order; we'll consider accumulator as second argument
- The meaning of the accumulator = the elements from the tree we already covered placed in an ordered list

Tree 2 list forward – code

% sort_1/3

% sort_1(in_tree,partial_result_list,final_result_list)

```
sort_1(nil,L,L).
sort_1(n(Left,Key,Right),Lin,Lout):-
     sort_1(Left,Lin,Llout),
     append(Llout,[Key],Lrin),
     sort_1(Right,Lrin,Lout).
```

Tree 2 list forward – code

```
sort_1(nil,L,L).
sort_1(n(Left,Key,Right),Lin,Lout):-
        sort_1(Left,Lin,Llout),
        append(Llout,[Key],Lrin),
        sort_1(Right,Lrin,Lout).
```

Clauses:

- When we reached the empty tree, the accumulator instantiate the result (default unification) as always in forward recursion.
- For the nonempty tree, the forward approach solves the parts of the partition in the standard order (first to last flow)
  - Solve the problem on the left subtree; the accumulator (content of the tree already covered, Lin) is passed as partial result. The result computed by this call will add all the keys in the left subtree to that accumulator and passed as accumulator (Llout) to the next processing task; sort_1(Left,Lin,Llout),
  - Solve the problem on the next part in the partition = Key; append(Llout,[Key],Lrin),
  - Solve the problem on the right subtree; the accumulator (content of the tree already covered, Lrin) is passed as partial result. The result computed by this call will add all the keys in the right subtree to that accumulator and passed as final result = on the whole tree); sort_1(Right,Lrin,Lout).

Tree 2 list forward – code analysis

Needs specific initial call:

```
sort_1(nil,L,L).
sort_1(n(Left,Key,Right),Lin,Lout):-
      sort_1(Left,Lin,Llout),
      append(Llout,[Key],Lrin),
      sort_1(Right,Lrin,Lout).

sort_1(Tree,Result):-
      sort_1(Tree,[],Result).
```

- Needs additional argument (accumulator)
- Evaluation:
- 2 recursive calls (a=2)
- 2 proportional parts (assuming tree is balanced; why is it a reasonable assumption?) (b=2)
  - Outside recursive call linear time (append) (c=1)
  - Hence, **O(nlgn)** – supralinear time, but a small quantity over n
- Please come back to this code AFTER we study difference lists

Tree 2 list backward – code

- Needs no additional argument
- The arguments needed are: input tree, output list,
- Order of arguments: input has to come first to benefit from the first argument indexation

```
% sort_2/2
% sort_2(in_tree, final_result_list)
sort_2(nil,[]).
sort_2(n(Left,Key,Right),Lout):-
     sort_2(Left,Llout),
     append(Llout,[Key],Lintout),
     sort_2(Right,Lrout),
     append(Lintout, Lrout,Lout).
```

Tree 2 list backward – code

```
sort_2(nil,[]).
sort_2(n(Left,Key,Right),Lout):-
        sort_2(Left,Llout),
        append(Llout,[Key],Lintout),
        sort_2(Right,Lrout),
        append(Lintout, Lrout,Lout).
```

Clauses:

- When we reached the empty tree, the result is empty.
- For the nonempty tree, the backward approach solves:
  - left subtree; the keys from left subtree generate the list from left out (Llout),
  - on the next part in the partition = Key to be added at the end of the Llout to create Lintout; append(Llout,[Key],Lintout),
  - right subtree; the keys from right subtree generate the list from right out (Lrout),
  - finally, the results of parts are concatenated (append(Lintout, Lrout,Lout)).

Tree 2 list backward– code analysis

```
sort_2(nil,[]).
sort_2(n(Left,Key,Right),Lout):-
      sort_2(Left,Llout),
      append(Llout,[Key],Lintout),
      sort_2(Right,Lrout),
      append(Lintout, Lrout,Lout).
```

- 2 recursive calls (a=2)
- 2 proportional parts (assuming tree is balanced) (b=2)
- Outside recursive call linear time (append) (c=1)
- Hence, **O(nlgn)** – supralinear time, but a small quantity over n
- Compared to the previous strategy?
- Time is larger (2 append calls instead of 1)

Tree 2 list backward– better?

- What can be done to improve?
- 2 append calls, both passing (almost) through the same list
- The first append go over the whole first argument (large list) in order to add just one element (Key) at the end.
- Better?
- What if somehow we rephrase:
  - Add at the end of the list
  - With add at the beginning of a list. How?
  - Instead of adding Key at the end of Llout
  - Add it at the beginning of Lrout
  - But is NOT available at this moment
  - Change order of processing

Tree 2 list backward – code

```
sort_3(nil,[]).
sort_3(n(Left,Key,Right),Lout):-
      sort_3(Left,Llout),
      sort_3(Right,Lrout),
      append(Llout,[Key| Lrout],Lout).
```

Clauses:

- When we reached the empty tree, the result is empty.
- For the nonempty tree, the backward approach solves:
  - left subtree; the keys from left subtree generate the list from left out (Llout),
  - right subtree; the keys from right subtree generate the list from right out (Lrout),
  - finally, the results of parts are concatenated with the unit element in between; just that between means now before the result on the right (append(Lintout, Lrout,Lout)).

Tree 2 list backward – code

```
sort_3(nil,[]).
sort_3(n(Left,Key,Right),Lout):-
      sort_3(Right,Lrout),
      Lint=[Key| Lrout],
      sort_3(Left,Llout),
      append(Llout, Lint,Lout).
```

Pure backward recursion code

Tree 2 list backward– code analysis

```
sort_3(nil,[]).
sort_3(n(Left,Key,Right),Lout):-
      sort_3(Left,Llout),
      sort_3(Right,Lrout),
      append(Llout,[Key| Lrout],Lout).
```

- 2 recursive calls (a=2)
- 2 proportional parts (assuming tree is balanced) (b=2)
- Outside recursive call linear time (append) (c=1)
- Hence, **O(nlgn)**
- Better (multiplicative constant) than the previous backward as just one append
- Better than forward – needs no additional parameter
- Can we do even better? Answer – we can! Check after difference lists