Logic Programming

Using Data Structures
Part 2

Temur Kutsia

Research Institute for Symbolic Computation
Johannes Kepler University of Linz, Austria
kutsia@risc.uni-linz.ac.at

Contents

Comparing Structures

Structure comparison:

- More complicated than the simple integers
- Have to compare all the individual components
- Break down components recursively.

Comparing Structures. `aless`

Example

`aless(X,Y)` succeeds if

- `X` and `Y` stand for atoms and
- `X` is alphabetically less than Y.

`aless(avocado,clergyman)` succeeds.
`aless(windmill,motorcar)` fails.
`aless(picture,picture)` fails.

Comparing Structures. `aless`

Success First word ends before second:
`aless(book,bookbinder).`

Success A character in the first is alphabetically less than one in the second:
`aless(avocado,clergyman).`

Recursion The first character is the same in both. Then have to check the rest:
For `aless(lazy,leather)` check
`aless(azy,eather).`

Failure Reach the end of both words at the same time:
`aless(apple,apple).`

Failure Run out of characters for the second word:
`aless(alphabetic,alp).`

Representation

- Transform atoms into a recursive structure.
- List of integers (ASCII codes).
- Use built-in predicate `name`:

```
?- name(alp,[97,108,112]).
yes

?- name(alp,X).
X = [97,108,112] ?
yes

?-name(X,[97,108,112]).
X = alp ?
yes
```

First Task

Convert atoms to lists:

```
name(X, XL).
name(Y,YL).
```

Compare the lists:

```
alessx(XL,YL).
```

Putting together:

```
aless(X,Y):-
    name(X,XL),
    name(Y,YL),
    alessx(XL,YL).
```

Second Task

Compose `alessx`.

Success First word ends before second:

`alessx([],[_|_]).`

Success A character in the first is alphabetically less than one in the second:

`alessx([X|_],[Y|_]:-X<Y.`

Recursion The first character is the same in both. Then have to check the rest:

`alessx([H|X],[H|Y]):-alessx(X,Y).`

What about failing cases?

Program

```
aless(X,Y):-
   name(X,XL),
   name(Y,YL),
   alessx(XL,YL).

alessx([],[_|_]).
alessx([X|_],[Y|_]:-X<Y.
alessx([H|X],[H|Y]):-alessx(X,Y).
```

Appending Two Lists

For any lists `List1`, `List2`, and `List3`
`List2` appended to `List1` is `List3` iff either

- `List1` is the empty list and `List3` is `List2`, or
- `List1` is a nonempty list and
  - the head of `List3` is the head of `List1` and
  - the tail of `List3` is `List2` appended to the tail of `List1`.

Program:

```
append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

Using append

Test	`?- append([a,b,c],[2,1],[a,b,c,2,1]).`
Total List	`?- append([a,b,c],[2,1],X).`
Isolate	`?- append(X,[2,1],[a,b,c,2,1]).`
	`?- append([a,b,c],X,[a,b,c,2,1]).`
Split	`?- append(X,Y,[a,b,c,2,1]).`

Inventory Example

Bicycle factory

- To build a bicycle we need to know which parts to draw from the supplies.
- Each part of a bicycle may have subparts.
- Task: Construct a tree-based database that will enable users to ask questions about which parts are required to build a part of bicycle.

Parts of a Bicycle

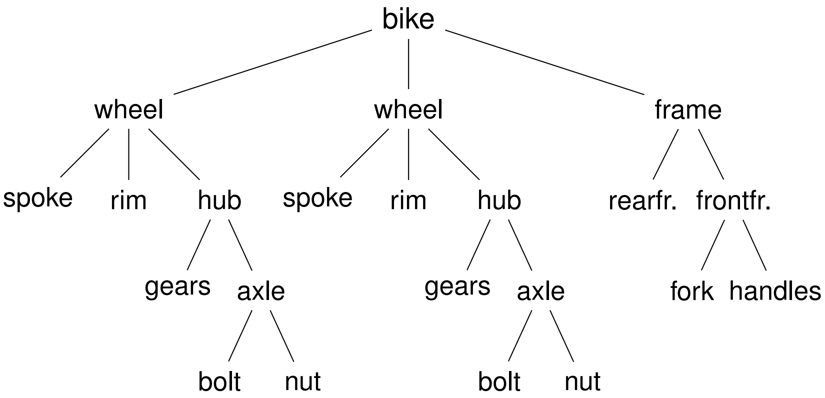
- Basic parts:

```
basicpart(rim).           basicpart(gears).
basicpart(spoke).         basicpart(bolt).
basicpart(rearframe).     basicpart(nut).
basicpart(handles).       basicpart(fork).
```

- Assemblies, consisting of a quantity of basic parts or other assemblies:

```
assembly(bike,[wheel,wheel,frame]).
assembly(wheel,[spoke,rim,hub]).
assembly(frame,[rearframe,frontframe]).
assembly(hub,[gears,axle]).
assembly(axle,[bolt,nut]).
assembly(frontframe,[fork,handles]).
```

Bike as a Tree



Program

Write a program that, given a part, will list all the basic parts required to construct it.

Idea:

1. If the part is a basic part then nothing more is required.
2. If the part is an assembly, apply the same process (of finding subparts) to each part of it.

Predicates: `partsof`

`partsof(X,Y)`: Succeeds if `X` is a part of bike, and `Y` is the list of basic parts required to construct `X`.

- Boundary condition. Basic part:

  ```
  partsof(X,[X]):-basicpart(X).
  ```

- Assembly:

  ```
  partsof(X,P):-
          assembly(X,Subparts),
          partsoflist(Subparts,P).
  ```

- Need to define `partsoflist`.

Predicates: `partsoflist`

- Boundary condition. List of parts for the empty list is empty:
  `partsoflist([],[]).`
- Recursive case. For a nonempty list, first find `partsof` of the head, then recursively call `partsoflist` on the tail of the list, and glue the obtained lists together:

```
partsoflist([P|Tail],Total):-
        partsof(P,Headparts),
        partsoflist(Tail,Tailparts),
        append(Headparts,Tailparts,Total).
```

▸ The same example using accumulators

Finding Parts

```
?- partsof(bike,Parts).

Parts=[spoke,rim,gears,bolt,nut,spoke,rim,
      gears,bolt,nut,rearframe,fork,handles] ;
No
```

```
?- partsof(wheel,Parts).

Parts=[spoke, rim, gears, bolt, nut] ;
No
```

Using Intermediate Results

Frequent situation:

- Traverse a PROLOG structure.
- Calculate the result which depends on what was found in the structure.
- At intermediate stages of the traversal there is an intermediate value for the result.

Common technique:

- Use an argument of the predicate to represent the "answer so far".
- This argument is called an accumulator.

Length of a List without Accumulators

Example

`listlen(L,N)` succeeds if the length of list `L` is `N`.

- Boundary condition. The empty list has length 0:
  ```
  listlen([],0).
  ```
- Recursive case. The length of a nonempty list is obtained by adding one to the length of the tail of the list.
  ```
  listlen([H|T],N):-
     listlen(T,N1),
     N is N1 + 1.
  ```

Length of a List with an Accumulator

Example

`listlenacc(L,A,N)` succeeds if the length of list `L`, when added the number `A`, is `N`.

- Boundary condition. For the empty list, the length is whatever has been accumulated so far, i.e. `A`:

```
lenacc([],A, A).
```

- Recursive case. For a nonempty list, add 1 to the accumulated amount given by `A`, and recur to the tail of the list with a new accumulator value `A1`:

```
lenacc([H|T],A,N):-
   A1 is A + 1,
   lenacc(T,A1,N).
```

Length of a List with an Accumulator, Cont.

Example

Complete program:

```
listlen(L,N):-lenacc(L,0,N).

lenacc([],A, A).
lenacc([H|T],A,N):-
   A1 is A + 1,
   lenacc(T,A1,N).
```

Computing List Length

Example (Version without Accumulator)

```
listlen([a,b,c],N).
listlen([b,c],N1), N is N1 + 1.
listlen([c],N2), N1 is N2 + 1, N is N1 + 1.
listlen([],N3), N2 is N3 + 1, N1 is N2 + 1, N
is N1 + 1.
N2 is 0 + 1, N1 is N2 + 1, N is N1 + 1.
N1 is 1 + 1, N is N1 + 1.
N is 2 + 1.

N = 3
```

Computing List Length

Example (Version with an Accumulator)

```
listlen([a,b,c],0,N).
listlen([b,c],1,N).
listlen([c],2,N).
listlen([],3,N).

N = 3
```

List as an Accumulator

- Accumulators need not be integers.
- If a list is to be produced as a result, an accumulator will hold a list produced so far.
- Wasteful joining of structures avoided.

Example (Reversing Lists)

```
reverse(List, Rev):-rev_acc(List,[],Rev).

rev_acc([],Acc,Acc).
rev_acc([X|T], Acc, Rev):-
  rev_acc(T,[X|Acc],Rev).
```

Bicycle Factory

Recall how parts of bike were found. Inventory example

`partsoflist` has to find the parts coming from the list `[wheel,wheel,frame]`:

- Find parts of `frame`.
- Append them to `[]` to find parts of `[frame]`.
- Find parts of `wheel`.
- Append them to the parts of `[frame]` to find parts of `[wheel,frame]`.
- Find parts of `wheel`.
- Append them to the parts of `[wheel,frame]` to find parts of `[wheel,wheel,frame]`.

Wasteful!

Bicycle Factory

Improvement idea: Get rid of append.

Use accumulators.

```
partsof(X,P):-partsacc(X,[],P).
partsacc(X,A,[X|A]):-basicpart(X).
partsacc(X,A,P):-
        assembly(X,Subparts),
        partsacclist(Subparts,A,P).
partsacclist([],A,A).
partsacclist([P|Tail],A,Total):-
        partsacc(P,A,Headparts),
        partsacclist(Tail,Headparts,Total).
```

`partsacc(X,A,P)`: parts of `X`, when added to `A`, give `P`.

Difference Structures

Compute parts of wheel without and with accumulator:

Example (Without Accumulator)

```
?- partsof(wheel,P).
X = [spoke, rim, gears, bolt, nut] ;
No
```

Example (With Accumulator)

```
?- partsof(wheel,P).
X = [nut, bolt, gears, rim, spoke] ;
No
```

Reversed order.

Difference Structures

How to avoid wasteful work and retain the original order at the same time?

Difference structures.

Difference Structures

Both accumulators and difference structures use two arguments to build the output structure.

Assumulators: the "result so far" and the "final result".

Difference structures: the "final result" and the "hole in the final result where the further information can be put".

Holes

- In a structure a hole is represented by a PROLOG variable which shares with a component somewhere in the structure.
- Example: `[a,b,c|X]` and `X`, a list together with a named "hole variable" where further information could be put.

Holes

Instantiating lists that contain a "hole":

1. Pass the "hole variable" as an argument to a PROLOG goal.
2. Instantiate this argument in the goal.
3. If we are interested in where further information can be inserted after this goal has succeeded, we will require this goal to pass back a new hole through another argument.

Holes

Example

Create a list with hole, add some elements in the list using the predicate `p` and then fill the remaining hole with the list `[z]`:

```
?- Res=[a,b|X], p(X, NewHole), NewHole=[z].
```

If our program contains a clause `p(H,H)`, then the goal returns `Res=[a,b,z]`.

If our program contains a clause `p([c|H],H)`, then the goal returns `Res=[a,b,c,z]`.

Bicycle Factory

Use holes.

```
partsof(X,P):-partshole(X,P,Hole),Hole=[].
partshole(X,[X|Hole],Hole):-basicpart(X).
partshole(X,P,Hole):-
        assembly(X,Subparts),
        partsholelist(Subparts,P,Hole).
partsholelist([],Hole,Hole).
partsholelist([P|Tail],Total,Hole):-
        partshole(P,Total,Hole1),
        partsholelist(Tail,Hole1,Hole).
```

Bicycle Factory. Detailed View

```
partsof(X,P):-partshole(X,P,Hole),Hole=[].
```

- `partshole(X,P,Hole)` builds the result in the second argument `P` and returns in `Hole` a variable.
- Since `partsof` calls `partshole` only once, it is necessary to terminate the difference list by instantiating `Hole` with `[]`. (Filling the hole.)
- Alternative definition of `partsof`:
  `partsof(X,P):-partshole(X,P,[]).`
  It ensures that the very last hole is filled with `[]` even before the list is constructed.

Bicycle Factory. Detailed View

```
partshole(X,[X|Hole],Hole):-basicpart(X).
```

- It returns a difference list containing the object (basic part) in the first argument.
- The hole remains open for further instantiations.

Bicycle Factory. Detailed View

```
partshole(X,P,Hole):-
        assembly(X,Subparts),
        partsholelist(Subparts,P,Hole).
```

- Finds the list of subparts.
- Delegates the traversal of the list to `partsholelist`.
- Two arguments `P` and `Hole` that make the difference list, are passed to `partsholelist`.

Bicycle Factory. Detailed View

```
partsholelist([P|Tail],Total,Hole):-
          partshole(P,Total,Hole1),
          partsholelist(Tail,Hole1,Hole).
```

- `partshole` starts building the `Total` list, partially filling it with the parts of `P`, and leaving a hole `Hole1` in it.
- `partsholelist` is called recursively on the `Tail`. It constructs the list `Hole1` partially, leaving a hole `Hole` in it.
- Since `Hole1` is shared between `partshole` and `partsholelist`, after getting instantiated in `partsholelist` it gets also instantiated in `partshole`.
- Therefore, at the end `Total` consists of the portion that `partshole` constructed, the portion of `Hole1` `partsholelist` constructed, and the hole `Hole`.