Smallest k sets revisited

– Prolog lab assignment 2 – Version 0.2α –
– D7012E Declarative languages –

Håkan Jonsson
Luleå University of Technology, Sweden

May 6, 2019

1 Assignment

Implement your solution to Haskell Lab Assignment 1 in Prolog. This was the assignment in which smallest k sets were computed. Run tests with the same examples.

2 Purpose

Why re-implement Haskell lab 1 in Prolog? To learn how to compute with lists and structures in Prolog. Apart from being of interest on its own, this is absolutely crucial to carry out Prolog Lab Assignment 3, where you will implement a board game complete with (ASCII) visualization, user interaction, and a clever computer opponent based on AI techniques.

2.1 Help with the implementation

Go back to your previous solution to Haskell lab 1, and freshen up your mind about how it works. It is not certain, but likely, that you can just translate many of the Haskell functions into Prolog procedures. However, Prolog does not function as Haskell does so you also need hands-on knowledge of Prolog before you start. Here is a good collection of informative problems *with* solutions:

`https://sites.google.com/site/prologsite/prolog-problems`

In particular, I recommend these problems in section 1 to start with:

- 1.01 (last), 1.02 (penultimate), 1.03 (k^{th} element), 1.04 (length), 1.05 (reverse), 1.06 (palindrome).
- 1.08 (remove duplicates), 1.09 (pack duplicates), 1.10 (run-length encoding), 1.12 (decode 1.10).
- 1.14 (duplicate), 1.15 (manifold).
- 1.20 (remove k^{th} element), 1.21 (insert), 1.22 (range).

Write the procedures yourself, even if you peek at the solutions. And, as always, if you google, you will find many many more Prolog problems.

As you will notice, you need to be even more careful in Prolog than in Haskell as there is (hardly) no type system around to help you. Simple spelling mistakes can cause very hard to find bugs. My advice is therefore that you write one procedure at a time and make sure it works before you start writing on the next procedure. Think and plan top-down. Implement and connect together bottom-up. I can not stress to much the need to be careful.

You could also write the "type" of a procedure as a comment above the first case (first line) of the function. Since there are no types in Prolog, use Haskell types or invent your own. The type should tell you what the parameters of the functions are. Example:

```
% append :: ([a], [a]) -> [a]
append([],L,L).
append([H|T],L2,[H|L3])  :-  append(T,L2,L3).
```

The first two parameters to `append` are inputs while the third is the result. My invented type, which is based on Haskell types, reminds me of this. Here, `[a]` means a list of some kind and `->` is supposed to say that what follows to the right is the result (the last parameter). Most likely, better types can be invented but this works for me. In the solutions to the problems above, another type notation is used.

2.1.1 Data structures

Lists are comfortable to use in Prolog. Depending on how you solved Haskell lab 1, you might need also structures for tuples. For instance, a sublist `[1,2,3]` with sum `6` starting at index `5` and ending at index `7` can be represented by the structure

```
sublist([1,2,3],6,5,7)
```

(We could also choose to name the functor something else than `sublist`.) A function that operates on such sublists can use a kind of pattern matching to take apart structures:

```
% sizeOf :: sublist([a],Int,Int,Int) -> Int
sizeOf(sublist(_,Size,_,_), Size).
```

Or, if the function operates on lists of sublists, we could do:

```
% totalSize :: [sublist([a],Int,Int,Int)] -> Int
totalSize([],0).
totalSize([sublist(_, Size,_,_)|T],S) :- totalSize(T,S2), S is Size+S2.
```