unige-principles-paradigms-programming-languages

by FedericoBozzini

The project

The project consists in solving the countdown problem where starting from a set of numbers (positive integers) the goal is to reach a target number (positive integer) by using only the four basic operation of addition, subtraction, multiplication and division in the least number of steps. All the intermediate results must be positive integers.

The solutions

General idea

The general idea behind the solution proposed is to recognise that the problem can be treated as a graph problem where the intermediate results are represented with nodes and the operations with the edges. The graph exploration starts from an initial state (0) and visits all the reachable states with only one operation. If the target number is not found, all the next states reachable in one step are visited next and evaluated. This process continues until a solution is found. To improve the efficiency a list of the visited states is kept, preventing the program from revisiting the already visited nodes of the graph. This methods exploits the lazy generation of new solutions, especially interesting in functional languages. A depth-first and a breadth-first solutions are also presented for completeness.

Haskell implementation

The Haskell implementation (*countdown.hs*) is especially simple due to the lazy evaluation strategy used by the language. In this case the solution is written with an infinite recursion and the language automatically evaluates only the necessary portion of the graph exploration until a valid solution is found.

Scala implementation

The Scala implementation (*Countdown.scala*) is very similar to the Haskell one. The only difference is due to the non-lazy nature of the language. To let the program evaluate only the minimum necessary "depth", the idea was to use the *Stream* data structure. A stream is evaluated only if necessary and reproduces the Haskell behavior.

Javascript implementation

An additional javascript implementation (*countdown.js*) was added, mainly to explore the implementation of a lazy solution in a language that is not functional (but offers several function constructs) and doesn't offer any ready-made tool to implement it. In this case the implementation is more complicated and uses a *generator*, a "low-level" construct, which requires the developer to manage more details.

As written before, also a depth-first and a breadth-first solutions are presented, mainly to check the correctness of the results and to evaluate the performances of the proposed solution.

Conclusions

Different languages offer different tools to solve a problem and the alghoritms may be easier or more difficult to implement on different languages. In the case of lazy evaluation, Haskell offers some very advanced features, other functional languages (Lisp, Scala, ...) offer some form of lazy evaluations (call-by-name, streams, thunks, ...) and other languages may have very basic or no support for it.