

FP Project:

An Evaluator in Haskell for a Logic Programming Language

In this project you have to write in Haskell evaluators for logic programming languages of various levels of complexity:

- The *propositional* case, i.e., every clause is (a combination of) elementary propositions (that is: predicates with zero variables).
- The case in which every clause is (a combination of) predicates with *one* variable.
- The general case, i.e., a predicate may have *any number* of variables.

For each case, the evaluator should be written as a Haskell function *eval* which gets as arguments a logic program (i.e., a list of clauses) and a query. The result of the *eval* function should be either *True* or *False*, or, in case the query contains one or more variables, a sequence of possible values for the variable(s) that make the query *true*.

The emphasis of this project is on the reasoning process in a logic programming language, i.e., you may assume that there are no side effects. Hence, you may ignore I/O. Also, arithmetic and lists do not have to be taken into account, i.e., the only terms you have to consider are *constants* and *variables*. Furthermore, no optimizations have to be investigated, so you may ignore the “!” from Prolog.

Deadline: Monday, June 6, 23:59. Submit your code on Blackboard under *FP Project group enroll*, together with a short explanation of your program.

Level 1 – Propositional Case – 3 points

In the propositional case an *atom* consists of a proposition letter only, i.e., atoms do not contain variables. Hence, unification and substitutions are not yet required in this part of the project. For example, the following is a small logic program in such a restricted language:

```
a0.  
a1.  
a2.  
  
b0 :- a0, a1.  
b1 :- a1, a2.  
b2 :- a1, a2, d.  
  
c0 :- b0, b1.  
c1 :- b0, b1, b2.
```

According to this program, the query $c_0?$ should give the answer *True*, whereas the query $c_1?$ should give *False* (because of the *closed world assumption*).

A possible starting point to program an evaluator in Haskell for Prolog programs like this might be given by data types like the following:

```

data Atom      =  A0 | A1 | A2 | B0 | B1 | B2 | C0 | C1 | D
                deriving (Eq, Show)

type Clause    =  (Atom, [Atom])

type Program   =  [Clause]

type Query     =  [Atom]

```

Elementary atoms are indicated by constructors in the algebraic type *Atom*. Note that constructors stand for *constants*, and start with a capital letter, which is different from the convention in Prolog.

The type *Clause* defines 2-tuples of the form (p, ps) , representing a clause in Prolog-syntax:

$$p \text{ :- } ps.$$

Even though an initial query is a single atom, during the process more atoms can be added. Hence, in general a query should be a list of atoms, as expressed by the type *Query*.

Task: Write a function *evalProp* which evaluates a query for the propositional case to *True* or *False*.

Level 2 – One-variable case – 5 points

In this case every atom consists of a *predicate* with *one* variable (or constant). The following program is an example of this case:

```

p(a).
p(b).
p(c).

q(a).
q(b).

r(X)  :-  p(X), q(X).

```

Note that an atom now consists of a *predicate* together with a *variable* or a *constant*. Now the queries $r(a)?$ and $r(b)?$ should yield *True*, whereas the query $r(c)?$ should give *False*. The query $r(X)?$ contains the variable X and should yield all values for X for which $r(X)$ is true. In this case, these values are a and b , whereas the value c is not correct.

In the above program, there is no immediate match for, e.g., $r(a)$, only after *substituting* a for X in the last clause we get the clause:

$$r(a) \text{ :- } p(a), q(a).$$

against which $r(a)$ can be matched. We remark that substitution may be done for variables *only*.

might be

type *Substitution*

Tasks:

- Extend or adapt the types given above such that atoms may consist of a predicate together with a single constant or variable.
- Define an operation \Leftarrow for substitution:

$$e \Leftarrow (x, a)$$

in which e is an “expression”, x a variable, and a a constant or a variable. The result should be that a is substituted for x in e .

Here, it should be possible that “expression” is a variable, a constant, or an atom. Hence, it is practical to define a type class such that the operation of substitution can be instantiated for several types.

- Since variables in a clause should be kept separate from the variables in the query, you should first write a function *rename* to replace the variables in the clause by new variables. Hence, you will have to generate *new* variables and substitute these for the variables in a clause.
- Write a function *unify* that finds the right substitution (in the form of a tuple (x, a)) which can make two atoms equivalent by applying that substitution. For example, the two atoms $p(a)$ and $p(x)$ (with a a constant and x a variable) can be *unified* by the substitution (x, a) :

$$\begin{aligned} p(a) \Leftarrow (x, a) &= p(a) && \text{substitution for } a \text{ leaves } a \text{ unchanged} \\ p(x) \Leftarrow (x, a) &= p(a) && \text{substituting } a \text{ for } x \text{ does replace } x \text{ with } a \end{aligned}$$

Note that neither $p(a)$ and $q(x)$, nor $p(a)$ and $p(b)$ can be unified.

- write a function *evalOne* that evaluates a query for a program in which clauses may consist of a predicate together with one variable or constant. Your function should work in case that the query contains a constant (yielding *True* or *False*), and also in the case that it contains a variable (yielding the substitutions for which the query becomes true).

Remark: again, the Prolog convention to let constants begin with a lowercase letter and a variable with a capital letter, does not need to bother us.

Level 3 – Multi-variable case – 2 points

This case is a generalization of the one-variable case, i.e., a predicate may have any number of variables. The example program on the royal family is an example of this case.

Tasks: The extra difficulty in this case is that an atom may contain the same variable more than once, such that substitution and unification become more tricky: in order to unify $p(x, x)$ and $p(a, y)$ both x and y have to be replaced by a .

Write a function *evalMulti* for this case. As before, your program should work for queries containing constants and/or variables.