Exercise (Week 8)

DUE: 30 July, 2019 14:00

CSE | **Stack**

Download the exercise tarball and extract it to a directory on your local machine. This tarball contains a file, called `Ex06.hs`, wherein you will do all of your programming.

To test your code, run the following shell commands to open a GHCi session:

```
$ stack repl
Configuring GHCi with the following packages: Ex06
Using main module: 1. Package 'Ex06' component exe:Ex06 ...
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Ex06             (Ex06.hs, interpreted)
Ok, one module loaded.
*Ex06> solutions e3
...
```

Note that you will only need to submit `Ex06.hs`, so only make changes to that file.

Logical formulas (often called constraints) ranging over finite domains (i.e., where variables are drawn from sets with a finite number of elements) are useful in a wide array of application areas ranging from solving planning problems, over compiler optimisations, to code verification. Given such formulas, we are usually interested in either whether there exists a solution (i.e. instantiation of all variables) that satisfy the logical formula or in the solutions themselves. You will have to perform the following tasks in this exercise:

- Write an evaluator for (unquantified) formula terms, using a GADT representation,

- check for the satisfiability of logical formulas, and
- compute the solution set of logical formulas.

We will do that for a very simple logic to keep it manageable, namely formulae of the form:

$$\exists x_1 \in S_1.\ \exists x_2 \in S_2.\ \ldots\ \exists x_n \in S_n.\ \phi$$

where ϕ is a term without any quantifiers and the sets $S_1 \ldots S_n$ are all finite.

Our language of quantifier-free terms is indexed by the type of the term: [1]

```
data Term t where
  Con     :: t -> Term t -- Constant values

  -- Logical operators
  And     :: Term Bool -> Term Bool -> Term Bool
  Or      :: Term Bool -> Term Bool -> Term Bool

  -- Comparison operators
  Smaller :: Term Int  -> Term Int  -> Term Bool

  -- Arithmetic operators
  Plus    :: Term Int  -> Term Int  -> Term Int
```

For example, the term `Con True` is of type `Term Bool`, and the term `Plus (Con 2) (Con 10)` is of type `Term Int`.

Our language of formulae (including quantifiers) is indexed by the types of each quantified variable:

```
data Formula ts where
  Body   :: Term Bool                         -> Formula ()
  Exists :: Show a
         => [a] -> (Term a -> Formula as) -> Formula (a, as)
```

For example, the following formula (`ex3` in the code):

```
Exists [False, True] $ \p ->
  Exists [0..2] $ \n ->
    Body $ p `Or` (Con 0 `Smaller` n)
```

Has the type `Formula (Bool, (Int, ()))` to reflect the types of the two quantified variables.

Evaluating Terms (2 marks)

Write a function:

```
eval :: Term t -> t
```

That recursively evaluates the term to its result.

Satisfiability check (3 marks)

Secondly, implement a function

```
satisfiable :: Formula ts -> Bool
```

that determines whether a given formula is satisfiable – i.e. whether there is an assignment of values to all existentially quantified variables (those with the `Exists` constructor), such that the body of the formula evaluates to `True`. Given the examples provided in the code, we expect:

```
*Ex06> satisfiable ex1
True
*Ex06> satisfiable ex2
True
*Ex06> satisfiable ex3
True
```

Enumerating Solutions (4 marks)

Finally, implement

```
solutions :: Formula ts -> [ts]
```

which computes a list of all the solutions for a `Formula`. Each individual solution is of the same type as the type index of the formula with one value for each existentially quantified variable. Again, considering the examples from `Formula.hs`, we expect:

```
*Ex06> solutions ex1
[()]
*Ex06> solutions ex2
[(1,()),(2,()),(3,()),(4,()),(5,()),(6,()),(7,()),(8,()),(9,()),(10,())]
*Ex06> solutions ex3
[(False,(1,())),(False,(2,())),(True,(0,())),(True,(1,())),(True,(2,()))]
```

What would be the result for a formula that is not satisfiable?

Note: You can use the *list monad* or list comprehensions here to make `solutions` a very short definition.

Submission instructions

```
$ give cs3141 Ex06 Ex06.hs
```

on a CSE terminal, or by using the `give` web interface. Your file *must* be named `Ex06.hs` (case-sensitive!). A dry-run test will *partially* autotest your solution at submission time. To get full marks, you will need to perform further testing yourself.

Footnotes:

[1] Note that there is also a `Name` constructor given in the code, however this constructor is **only** used for pretty printing and is not relevant to any of the functions you have to implement.