185.A05 Advanced Functional Programming SS 2020

Friday, 29 May 2020

Assignment 7

on Chapter 16 (and chapters on higher-order function computing)

Topics: Logical programming functionally, combinator programming, higher-order functions programming

Submission deadline: Monday, 8 June 2020, 12am

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Important:

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 1. If you have any questions regarding these instructions, ask your questions in the TISS forum. Following these instructions is paramount to ensure a smooth processing of your submitted file with the test system.

2. Store all functions to be written for this assignment in a top-level file named

   `Assignment7.hs`

   of your group directory starting with the module declaration

   ```
   module Assignment7 where
   ```

   Comment your program meaningfully; use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 7.

3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment7.hs`. Import declarations for self-defined modules will fail: Only `Assignment7.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.

4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Programming tasks:

1. Implement the combinator library of Chapter 16 (note: the file accessible via the link in column ‘Remarks’ on the webpage of the course contains major parts of the code of Chapter 16).

2. Add resp. complete missing implementations of functions, especially the instance declarations for the types `Term`, `Subst`, and `Answer` for the type class `Show` in order to render possible outputs as shown in the examples of Chapter 16 and below.

3. **Without submission:** Test and validate your implementation by applying the predicates `append` and `good` of Chapter 16 using calls like the ones below and additional ones of your own choice:

```
run (append (list [1,2], list [3,4], var "z")) :: Stream Answer
 ->> [{z=[1,2,3,4]}]
run (append (var "x", var "y", list [1,2,3])) :: Stream Answer
 ->> [{x = Nil, y = [1,2,3]},
      {x = [1], y = [2,3]},
      {x = [1,2], y = [3]},
      {x = [1,2,3], y = Nil}]
run (append (var "x", list [2,3], list [1,2,3])) :: Stream Answer
 ->> [{x = [1]}]
run (good (list [1,0,1,1,0,0,1,0,0])) :: Stream Answer
 ->> [{}]
run (good (list [1,0,1,1,0,0,1,0,1])) :: Stream Answer
 ->> []
run (good (var "s")) :: Stream Answer
  ->>[{s=[0]},
      {s=[1,0,0]},
      {s=[1,0,1,0,0]},
      {s=[1,0,1,0,1,0,0]},
      {s=[1,0,1,0,1,0,1,0,0]}

run (good (var "s")) :: Diag Answer
  ->> Diag [{s=[0]},
            {s=[1,0,0]},
            {s=[1,0,1,0,0]},
            {s=[1,0,1,0,1,0,0]},
            {s=[1,1,0,0,0]},
            {s=[1,0,1,0,1,0,1,0,0]},
            {s=[1,1,0,0,1,0,0]},
            {s=[1,0,1,1,0,0,0]},
            {s=[1,1,0,0,1,0,1,0,0]}
run (good (var "s")) :: Matrix Answer
 ->>MkMatrix [[],
       [{s=[0]}],[],[],[],
       [{s=[1,0,0]}],[],[],[],
       [{s=[1,0,1,0,0]}],[],
       [{s=[1,1,0,0,0]}],[],
       [{s=[1,0,1,0,1,0,0]}],[],
       [{s=[1,0,1,1,0,0,0]},{s=[1,1,0,0,1,0,0]}],[], (usw.)
```

4. Referring to the implementation of `append` of Chapter 16 for inspiration, implement a predicate `shuffleLP` (with 'LP' reminding to logical programming):

```
shuffleLP :: Bunch m => (Term, Term, Term) -> Pred m
shuffleLP (p,q,r) = ...
```

   For `Int` lists, `shuffleLP` shall behave like `shuffle` below, however, the distinction between input and output variables shall be abolished (like for `append`):

```
shuffle :: [a] -> [a] -> [a]
shuffle [] ys     = ys
shuffle (x:xs) ys = x : shuffle ys xs
```

5. **Without submission:** Test and validate your implementation of `shuffleLP` with appropriate calls of your own choice, e.g.:

```
run (shuffleLP (list [1,2,3], list [4,5,6], var "z")) :: Stream Answer
 ->> [{z=[1,4,2,5,3,6]}]
run (shuffleLP (var "x", list [4,5,6], list [1,4,2,5,3,6])) :: Stream Answer
 ->> [{x=[1,2,3]}]
run (shuffleLP (var "x", var "y", list [1,3,2,4])) :: Stream Answer
 ->> ...many values possible for x and y
```

   Additionally, investigate the impact of using a different search monad (i.e., replacing `Stream Answer` by `Diag Answer` or `Matrix Answer`) on the output.

6. Finally, we consider so-called proper sequences inductively constructed from the one-elements lists [0] and [1]. We define:

   1. *Atoms*: The sequences [0] and [1] are proper.
   2. *Composition*: If s_1 and s_2 are proper sequences, then the sequence $s_1 + +[2] + +s_2$ is a proper sequence, too.
   3. *Parentheses*: If s is a proper sequence, then the sequence $[3] + +s + +[4]$ is a proper sequence, too.
   4. Rules 1 to 3 define all proper sequences; there are no other ones.

   Following the implementation of the predicate `good` of Chapter 16, implement a predicate `proper` of type:

```
proper :: Bunch m => Term -> Pred m
proper (s) = ...
```

   for recognizing and generating proper sequences.

7. **Without submission:** Test and validate your implementation of `proper` with like those below and additional ones of your own choice:

```
run (proper (list [3,0,2,1,4,2,3,1,4])) :: Stream Answer
 ->> [{}]          (w/ the meaning: Argument was proper)
run (proper (list [3,1,2,1,3])) :: Stream Answer
 ->> []      (w/ the meaning: Argument was not proper)
run (proper (var "r")) :: Stream Answer
 ->> ...
run (proper (var "r")) :: Diag Answer
 ->> ...
run (proper (var "r")) :: Matrix Answer
 ->> ...
```

8. Last but not least, we reconsider the programming language MINI of assignment 6. Identifying the type names of the abstract syntax representation of MINI programs of assignment 6 with the set of all programs, statements, expressions, etc.:

P :	**PROG**	(set of all) programs
S :	**STMT**	(set of all) statements
E :	**EXPR**	(set of all) arithmetic expressions
PE :	**PEXPR**	(set of all) predicate expressions
V :	**IDF**	(set of all) (variable) identifiers
I :	**INT**	(set of all) integer representations as digit sequences
F :	**FLOAT**	(set of all) float representations as two digit sequences separated by '.'

and introducing additionally the notions:

State	(set of all) program states (i.e., maps from (variable) identifiers to values)
Val	(set of all) expression values (i.e., integer and float values)
Boolean	set of Boolean values (i.e., $True$ and $False$)
Id_{State}	Identity function on the set of states defined by: $Id_{State} = \lambda\,\sigma.\ \sigma$

we can define a semantics $\mathcal{S}$ for the abstract syntax trees of well-formed MINI programs:

$$\mathcal{P} : \mathbf{PROG} \to (\mathbf{State} \to \mathbf{State})$$
$$\mathcal{P}[\![\, S \,]\!](\sigma) = \mathcal{S}[\![\, S \,]\!](\sigma)$$

$$\mathcal{S} : \mathbf{STMT} \to (\mathbf{State} \to \mathbf{State})$$
$$\mathcal{S}[\![\, [\,] \,]\!] = \lambda\,\sigma.\ \sigma$$
$$\mathcal{S}[\![\, S_1 : [S_2] \,]\!] = \mathcal{S}[\![\, [S_2] \,]\!] \circ \mathcal{S}[\![\, S_1 \,]\!]$$
$$\mathcal{S}[\![\, \texttt{Ass}\ E_1\ E_2 \,]\!] = \lambda\,\sigma.\ \sigma[\mathcal{E}[\![\, E_2 \,]\!](\sigma)/E_1]$$ (Semantic substitution, E_1 must represent a variable)

$$\mathcal{S}[\![\ \texttt{If}\ PE\ S_1\ S_2\ ]\!](\sigma) = \begin{cases} \mathcal{S}[\![\ S_1\ ]\!](\sigma) & \text{if } \mathcal{PE}[\![\ PE\ ]\!](\sigma) = True \\ \mathcal{S}[\![\ S_2\ ]\!](\sigma) & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\ \texttt{While}\ PE\ S\ ]\!](\sigma) = \begin{cases} (\mathcal{S}[\![\ \texttt{While}\ PE\ S\ ]\!] \circ \mathcal{S}[\![\ S\ ]\!])(\sigma) & \text{if } \mathcal{PE}[\![\ PE\ ]\!](\sigma) = True \\ \sigma & \text{otherwise} \end{cases}$$

$$\mathcal{S}[\![\ \texttt{Repeat}\ S\ PE\ ]\!](\sigma) = \begin{cases} \mathcal{S}[\![\ S\ ]\!](\sigma) & \text{if } \mathcal{PE}[\![\ PE\ ]\!](\mathcal{S}[\![\ S\ ]\!](\sigma)) = True \\ (\mathcal{S}[\![\ \texttt{Repeat}\ PE\ S\ ]\!] \circ \mathcal{S}[\![\ S\ ]\!])(\sigma) & \text{otherwise} \end{cases}$$

$$\mathcal{E} : \mathbf{EXPR} \to (\mathbf{State} \to \mathbf{Val})$$

$$\begin{aligned} \mathcal{E}[\![\ \texttt{I}\ i\ ]\!](\sigma) &= \mathit{NaturalInterpretation}(i) \\ \mathcal{E}[\![\ \texttt{F}\ f\ ]\!](\sigma) &= \mathit{NaturalInterpretation}(f) \\ \mathcal{E}[\![\ \texttt{V}\ s\ ]\!](\sigma) &= \sigma(s) \\ \mathcal{E}[\![\ \texttt{Plu}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{plus}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \\ \mathcal{E}[\![\ \texttt{Min}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{minus}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \\ \mathcal{E}[\![\ \texttt{Mul}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{times}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \\ \mathcal{E}[\![\ \texttt{Div}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{divide}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \end{aligned}$$

$$\mathcal{PE} : \mathbf{PEXPR} \to (\mathbf{State} \to \mathbf{Boolean})$$

$$\begin{aligned} \mathcal{PE}[\![\ \texttt{Equal}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{equal}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \\ \mathcal{PE}[\![\ \texttt{NEqual}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{nequal}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \\ \mathcal{PE}[\![\ \texttt{GEqual}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{greater}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \\ \mathcal{PE}[\![\ \texttt{LEqual}\ E_1\ E_2\ ]\!](\sigma) &= \mathit{smaller}(\mathcal{E}[\![\ E_1\ ]\!](\sigma), \mathcal{E}[\![\ E_2\ ]\!](\sigma)) \end{aligned}$$

where *equal*, *nequal*, *greater*, *smaller* denote the equality, inequality, greater than, and smaller than relations on integers and floats, and *plus*, *minus*, *times*, *divide* the addition, substraction, multiplication, and divide operations on integers and floats. Operations and relations involving two integer arguments resp. two float arguments are supposed to be the standard operations and relations on integers and floats, respectively. Opperations involving both an integer and a float argument are supposed to cast the integer argument to its corresponding float value and to work on floats. Last but not least, *NaturalInterpretation* interpretes a digit sequence (possibly proceeded by the sign symbol - and splitted by a '.') in the 'natural sense' as the integer resp. float it represents.

8.1 Implement a Haskell function `interpreter`, which, applied to the abstract syntax tree of a well-formed MINI program p and an initial program state σ computes the state p is terminating in if applied to σ. If p is not well-formed, or the initial state not defined for all variable identifiers referred to in p, no specific behaviour of `interpreter` is required. Note, if an arithmetic operator or a comparison involves both integer and float values and or integer and float valued variables, an automatic type cast for integer values is to be performed.

```
type Identifier   = String
type Program_Name = Identifier
type Variable     = Identifier
```

```
type PN           = Program_Name

data P  = P PN [S] deriving (Eq,Show)    -- P for program
data S  = Ass E E                        -- S for statement
          | If PE [S] [S]
          | While PE [S]
          | Repeat [S] PE deriving (Eq,Show)
data E  = I Integer                      -- E for expression
          | F Float
          | V Variable
          | Plu E E
          | Min E E
          | Mul E E
          | Div E E deriving (Eq,Show)
data PE = Equal E E           -- PE for predicate expression
          | NEqual E E
          | GEqual E E
          | LEqual deriving (Eq,Show)

type State = (Variable -> Either Int Float)

interpreter :: P -> State -> State
```

8.2 Obvioulsy, abstract syntax trees suitable as input for `interpreter` may result from parsing well-formed MINI programs using the combinator or monadic parser of MINI programs of assignment 6. Thus, copy and paste your preferred parser of assignment 6 to submission file `Assignment7.hs` and use it to implement an interpreter that directly starts from a (well-formed) MINI program. If the argument program is not well-formed or the argument state not defined for all variable identifiers referred to in the program, no specific behaviour of `mini_interpreter` is required.

```
type Mini_Program = String

mini_interpreter :: Mini_Program -> State -> State
mini_interpreter mp sigma = interpreter...
```

8.3 **Without submission:** Test and validate `mini_interpreter` by means of a sample of MINI programs of your choice, e.g., write MINI programs computing the factorial or Fibonacci function.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller