Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Functional Programming

WS 2019/20

Benjamin Dietrich, Denis Hirn

Assignment #7

Submission Deadline: Thu, 12.12.2019

Wichtig: Im Zeitraum vom 6.12.2019 bis zum 19.12.2019 (16 Uhr) findet die Lehrevaluation zu dieser Veranstaltung statt. Sie wurden dazu per E-Mail an Ihre studentische Adresse eingeladen. Bitte nehmen Sie daran teil, das Feedback ist uns sehr wichtig!

Exercise 1: Deep Embedding of a While-Language (10 Points)

In the lecture we have implemented a deeply embedded, typed expression language over Integers and Booleans. To ensure that only well-typed expressions can be built, the module makes use of Haskell's *Generalized Algebraic Data Types* (`GADTs`). GADT syntax allows us to check the typability of our embedded language at compile time. In this exercise we are going to extend this DSL to a simple *While-Language*.

Inspect your repository to find the module `DeepWhile` which contains a slightly modified version of the lecture's code to start with.

The given module exports the AST data type `Expr a`, together with its constructors and an observer function `eval :: Expr a -> a` which evaluates DSL expressions.

1. First, extend the expression language with a relational operator `Lt` which takes two expressions e_1, e_2 of type `Expr Int` and – when evaluated – returns whether $e_1 < e_2$ or not. Complete the `Show` instance and `eval` observer accordingly.

   Example:

   ```
   eval (Lt (ValI 41) (ValI 42)) ≡ True
   ```

2. Extend the expression language with support for a *variable environment* (see type `Env b` in `DeepWhile`[1]) which binds values of a particular type `b` to variable names (type `String`) that can be used in the DSL expression:

   (a) Equip the data type `Expr` with a second type variable `b` that determines the type of environment values (`Expr a b`: evaluates to a value of type `a`, based on an environment of variables of type `b`).

   (b) Adapt `eval` to additionally take an environment of type `Env b`:

[1] The environment is implemented as a dictionary using the module `Data.Map`. See https://hackage.haskell.org/package/containers-0.6.2.1/docs/Data-Map-Lazy.html for further information.

```
eval :: Env b -> Expr a b -> a
```

(c) Provide an additional constructor `Var` which takes a variable name (`String`). Var represents a lookup of the variable name in the environment on evaluation. If the variable is not bound in the environment an `error "Variable not bound!"` is thrown at runtime. Complete `Show` and `eval` accordingly.

Example:

```
eval (M.singleton "a" 42) (Var "a") ≡ 42
```

3. Define a new data type `Stmt b` for While-Language statements operating on an environment of type `b`. Use GADT syntax to define the following constructors:

   (a) `Assign` – takes a variable name (`String`) and an expression (`Expr`) that evaluates to a value of type `b`. It represents the assignment of the expression result to a variable with the given name.

   (b) `While` – takes a Boolean expression and a statement (`Stmt b`) which is to be repeated while the condition holds. Both, the expression and the loop statement are evaluated based on an environment of type `b`.

   (c) `Seq` – takes two statements (`Stmt b`) and represents their sequential execution.

   Define a `Show` instance for `(Stmt b)`.

4. Write a function

```
runS :: Env b -> Stmt b -> Env b
```

which runs a given statement based on a given environment. The – possibly modified – environment is returned. Whenever statements are run sequentially (or looped) the modifications in the environment of the previous statement are visible in the execution of the subsequent.

Example:

```
let p = Seq (Assign "a" $ ValI (21)) $
        Seq (Assign "r" $ ValI 0) $
             While (Lt (Var "r") (Var "a")) $
               Seq (Assign "a" $ Add (Var "a") (ValI 1)) $
                    Assign "r" $ Add (Var "r") (ValI 2)


runS M.empty p ≡ M.fromList [("a",42),("r",42)]
```

5. Finally, define and export a function

```
run :: String -> Stmt b -> Maybe b
```

which takes a variable name v and a statement s. This function should run s with an empty environment and returns the value bound to v after evaluation (or `Nothing` if v was not bound by the program).

Example:

```
run p ≡ 42
```

Exercise 2: Mathematical Expressions with Only Four 4s **(10 Points)**

Solve the following puzzle:

> For each number n between 0 and 20, find at least one mathematical expression which evaluates to n and is an arbitrary combination of exactly four numbers 4 using the following arithmetic operations: addition ($a+b$), subtraction ($a-b$), multiplication ($a*b$), division (a/b), exponentiation (a^b), square root ($\sqrt{a}$) and factorial of numbers (4!).

Example:

$$0 = \frac{4}{4} * 4 - 4 \qquad 1 = \left(\frac{4}{4}\right)^{4^4} \qquad \cdots$$

1. Define a data type `ExprTree` to represent such mathematical expressions.
2. Write a function `eval :: ExprTree -> Maybe Float` which evaluates an expression to a number if possible; if the expression is not evaluable – due to division by zero – return `Nothing`, instead.
3. Write a function `trees :: [ExprTree] -> [ExprTree]` which takes a list of leaf nodes and returns a list of all expression trees that can be built in combination of these leaf nodes. Assume that a sequential application of the unary square-root-operation (*e.g.* $\sqrt{\sqrt{a}}$) is not allowed, while factorial is only applied to leaf nodes (4!), but not to other expressions (*e.g.* $(a+b)!$).

   Example: (Note: This example uses a *human-digestable* notation for **ExprTree** values.)

   ```
   trees [4,4,4,4] => [ (4+4+4+4), ...
                      , (4*4+4+4), ...
                      , (4*(4+4)+4), ...
                      , ((4/4)^4)^4, ...
                      , ((4!+4)/4+4), ...
                      ]
   ```

   Hint: It might be useful to write a helper function `splits :: [a] -> [([a], [a])]` which returns all combinations of a list split in two parts.
   Example: `splits [1..4] => [([1],[2,3,4]),([1,2],[3,4]),([1,2,3],[4])]`

4. Finally, write a function `solution :: Int -> Maybe ExprTree` that returns one arbitrary "expression of four 4s" which evaluates to a given number $n \in \{1, \ldots, 20\}$. Return `Nothing` if no such expression was found.