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 #11

Submission Deadline: Thu, 30.1.2020

Exercise 1: List Comprehensions (8 Points)

List comprehensions can systematically be *desugared* to Haskell code using only standard list functions `concat` and `map`.[1]

1. For each of the following list comprehensions, define a semantically equivalent version that uses neither list comprehensions nor explicit recursion, but only `Prelude` list functions `concat` and `map` (you may also use `filter` here).

   (a) `c1 a b = [ y | x <- [a..b], y <- [x-1, x, x+1] ]`

   (b) `c2 = [ (a, b, c) | a <- [1..100], b <- [1..100], c <- [1..100], a^2 + b^2 == c^2 ]`

   (c) `c3 xss = [ x | xs <- xss, length xs == 2, x <- xs ]`

   (d) `c4 = [ (x, [ y | y <- [5..8], x + y > 8 ] ) | x <- [1..4] ]`

If we have a look at the `Monad` instance of `[]` again, we can see that the bind operator is defined in terms of `concat` and `map`, too:

```
instance Monad [] where
  return x = [x]
  xs >>= g = concat (map g xs)
```

2. For each of the definitions in 1, find a semantically equivalent version that does neither use list comprehensions nor explicit recursion nor standard list functions `concat` or `map`, but only the monadic combinators `(>>=)` and `return` of the list monad instance. (You may also use do-notation instead).

[1] See the introductive thread in the forum: https://forum-db.informatik.uni-tuebingen.de/t/list-comprehensions/7422/3

Exercise 2: Monadic Evaluation of Expressions (12 Points)

Once again, expressions need to be evaluated. Consider the following (incomplete) type of expressions over integers and Booleans. The complete code can be found in module `Eval`.

```
data IntOp = Mul | Div | Eq

data Val = IntV Int | BoolV Bool

data BoolOp = And | Or

data Expr = Lit Val
          | AppInt IntOp Expr Expr
          | AppBool BoolOp Expr Expr
          | Var String
          | Let (String, Expr) Expr
```

The language supports *let*-bindings (`Let`) and variable references (`Var`). In a term `Let [("x",` e_1`)]` e_2, expression e_2 is evaluated in an **environment** in which the name x is bound to the result of expression e_1. The following term represents the computation 42^2.

```
Let ("x", Lit $ IntV 42) (AppInt Mul (Var "x") (Var "x"))
```

During evaluation of terms, a number of things might go wrong, causing the evaluation to **fail**. We might encounter division by zero, ill-typed expressions (*e.g.* `Mul` applied to Booleans) or references to variables which have not been bound by an enclosing `Let`.

We could define an evaluation function which passes an environment explicitly and wraps its result in `Maybe` to account for failing evaluations. However, doing all this manually quickly proves tedious and verbose. Instead, we will define an *evaluation monad* which provides for both requirements (passing an environment and possibility to fail).

An environment `type Env = [(String, Val)]` is a mapping from variable names to values. An evaluation of expressions (`Expr`) in such an environment can be modeled by a function of type `Env -> Val`, *i.e.* a function that receives an environment and returns a value, which is based on the variable bindings of the environment. Since the evaluation in the environment might fail, we enhance the return type to `Env -> Maybe Val`.

Extending the idea to any evaluations in the context of an environment (not only of return type `Val`), we define the data type `Eval` as follows:

```
-- | Evaluation in an environment.
newtype Eval a = E { runEval :: Env -> Maybe a }
```

1. Before approaching the monadic aspect, we will define a few helper functions for expression evaluation.

   (a) Define a function `lookupEnv :: String -> Eval Val` that looks up a variable name in the environment.

   Example:

   ```
   > runEval (lookupEnv "x") [("x", IntV 1)]
   Just (IntV 1)
   ```

(b) Define a function `local :: (Env -> Env) -> Eval Val -> Eval Val` that evaluates an expression in a modified environment. `local f m` modifies the environment using function `f` and runs the expression evaluation `m` in the modified environment.

Example:

```
> runEval (local (const [("x", IntV 2)]) $ lookupEnv "x") [("x", IntV 1)]
Just (IntV 2)
```

2. Now the interesting part: Make the type `Eval` an instance of `Monad`, *i.e.*

```
instance Monad Eval where
    return = ...
    (>>=)  = ...
```

Implementing the `(>>=)` method is not trivial. You should expect to need some time to figure out how things need to be combined. However, the rest of the assignment will then be straightforward.

3. We now have everything in place to actually evaluate expressions in the `Eval` monad. Define the core evaluation function `eval :: Expr -> Eval Val`. Also, define a helper function `evalTop :: Expr -> Maybe Val` that evaluates an expression in an empty initial environment using `eval`.

   Hint: Use `local` to modify the environment on `Let` bindings.