Assignment 9

2019-12-11 15:44
08d925d

due *Tue 12 Nov*

For this assignment, revise the toy calculator language from November 6 (see N20191106.hs on gitlab) to support the following features.

1. Add a constructor `DivisionByZero` to the `Error` enumeration type. Revise the `interpret` function so that it reports this error when needed, rather than calling the Haskell division operator (which produces `Infinity`). Here are some examples. Note that the first includes `Num 0` as a constant, but the second is more subtle because a sub-tree evaluates to zero.

   ```
   λ> interpret defaultEnv (BinaryOp Div (Var "x") (Num 0))
   Left DivisionByZero
   λ> interpret defaultEnv (BinaryOp Div (Var "z") (BinaryOp Sub (Var "x") (Var
   Left DivisionByZero
   ```

2. Create a function with the signature

   ```
   optimize :: Absyn -> Absyn
   ```

   Its purpose will be to *simplify* syntax trees so that there is less work to do when it comes time to interpret them. Here are some arithmetic identities that the optimize function should implement:

   ```
   e + 0 == e
   0 + e == e
   e * 1 == e
   1 * e == e
   e / 1 == e
   e - 0 == e
   0 - e == -e  (Switch from subtraction to negation)
   ```

   In those equations, `e` refers to any expression – it does not need to be a variable, but could be a sub-tree.

   Also, whenever all the operands of a unary or binary operator node are numeric constants (or themselves *optimize* to constants), it will just evaluate them right away, replacing them with a constant. Therefore, the optimizer is also *partially* an interpreter. What keeps it from being the same as the full interpreter is that it doesn't have access to the environment (so calculations with variables cannot be evaluated) and it can't report an error (so it should never divide by zero). Here are some sample results:

   ```
   λ> optimize (UnaryOp Sqrt (BinaryOp Add (Num 4) (Num 5)))
   ```

```
Num 3.0
λ> optimize (UnaryOp Sqrt (BinaryOp Add (Var "k") (Num 0)))
UnaryOp Sqrt (Var "k")
λ> optimize (UnaryOp Negate (BinaryOp Mult (Num 1) (Var "t")))
UnaryOp Negate (Var "t")
λ> optimize (BinaryOp Sub (Num 0) (BinaryOp Mult (Num 1) (Var "t")))
UnaryOp Negate (Var "t")
λ> optimize (Local "n" (UnaryOp Sqrt (Num 16)) (BinaryOp Div (Var "n") (N
Local "n" (Num 4.0) (Var "n")
λ> optimize (Local "n" (UnaryOp Sqrt (Num 16)) (BinaryOp Div (Var "n") (N
Local "n" (Num 4.0) (BinaryOp Div (Var "n") (Num 0.0))
```