Expression Tree Calculator Case Study

H. Conrad Cunningham

13 September 2018

Contents

H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of September 2018 is a recent version of Firefox from Mozilla.

Expression Tree Calculator Case Study

Problem Description

In programming, we often use trees and other hierarchical data structures.

We can illustrate how to implement a tree in Haskell using a small calculator program for simple arithmetic expressions composed of addition operations, integer constants, and variables. Examples of such expressions in infix form area `1+2` and `(x+x)+(7+y)`.

We can represent expressions naturally with a tree, where nodes are operations (e.g., addition) and leaves are values (e.g., constants or variables). This representation is called the *abstract syntax tree* for the expression.

In Haskell, we can represent these expression trees using algebraic data types. Such types often enable us to express programs concisely by using pattern matching.

For the calculator program, we introduce the following types to describe the expression tree.

```
type Name = String

data ExprTree = Add ExprTree ExprTree |
                Var Name |
                Val Int
                deriving Show
```

Above `Add` represents addition of two subexpressions, `Var` represents a variable with a name, and `Val` represents a constant value.

Consider a function to evaluate an expression in some *environment*. The purpose of an environment is to associate values with variables.

For example, the expression `x+1` might be evaluated in an environment that associates the value `5` with the variable `x`, written `{ x -> 5 }`. This evaluation yields the value `6`.

An environment associates a variable name with a value. The environment `{ x -> 5 }` given above can be expressed in Haskell in a number of ways. Here we choose to represent it as an *association list*, that is, as a list of pairs where the variable is the first component and its value is the second:

```
[("x",5)]
```

To simplify our evaluation program, we define the type synonym `Env` as follows:

```
type Env = [(Name,Int)]
```

We can use the Prelude function `lookup` to search association lists. It takes a `key` and an association list and returns the value associated with the key, if any. It wraps the result in a `Maybe`, returning a `Just` if the key is found or returns a `Nothing` if it does not occur in the list.

```
lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup _   []   =  Nothing
lookup key ((x,y):xys)
    | key == x  =  Just y
    | otherwise =  lookup key xys
```

We can now define the evaluation function in Haskell as follows:

```
eval :: ExprTree -> Env -> Int
eval (Add l r) env = eval l env + eval r env
eval (Var n)   env =
    case (lookup n env) of
        Just i  -> i
        Nothing -> error ("Undefined variable " ++ show n)
eval (Val v)   _   = v
```

To explore algebraic data types and pattern matching further, consider another operation on arithmetic expressions: symbolic derivation. Looking back at our calculus class, we see the following rules for differentiation:

- The derivative of a sum is the sum of the derivatives.
- The derivative of some variable `v` is 1 if `v` is the variable relative to which the derivation takes place, and is 0 otherwise.
- The derivative of a constant is 0.

We can directly translate these rules into a Haskell function that uses the above data types as follows:

```
derive :: ExprTree -> Name -> ExprTree
derive (Add l r) v = Add (derive l v) (derive r v)
derive (Var n)   v
    | v == n       = Val 1
derive _         _ = Val 0
```

Consider an example with a simple `main` function that performs several operations on the expression `(x+x)+(7+y)`.

```
main = do
    let exp = Add (Add (Var "x") (Var "x"))
                  (Add (Val 7) (Var "y"))
    let env = [("x",5), ("y",7)]
    putStrLn ("Expression: " ++ show exp)
    putStrLn ("Evaluation with x=5, y=7: " ++
              show (eval exp env))
    putStrLn ("Derivative relative to x:\n " ++
        show (derive exp "x"))
    putStrLn ("Derivative relative to y:\n " ++
              show (derive exp "y"))
```

It first computes its value in the environment `{ x -> 5, y -> 7 }` and then computes its derivative relative to `x` and then to `y`.

Executing this program, we get the expected output:

```
Expression: Add (Add (Var "x") (Var "x")) (Add (Val 7) (Var "y"))
Evaluation with x=5, y=7: 24
Derivative relative to x:
```

```
    Add (Add (Val 1) (Val 1)) (Add (Val 0) (Val 0))
Derivative relative to y:
    Add (Add (Val 0) (Val 0)) (Add (Val 0) (Val 1))
```

The result of the derivative is complex. It should be simplified before printing. Defining a basic simplification function using pattern matching is an interesting (but surprisingly tricky) problem.

Here is an skeleton function that simplifies the expression by evaluating constant subexpressions and accounting for identity elements.

```
simplify :: ExprTree -> ExprTree
simplify t@(Val _)               = t
simplify t@(Var _)               = t
simplify (Add (Val 0) r        ) = simplify r
simplify (Add l         (Val 0)) = simplify l
simplify (Add (Val x) (Val y))   = Val (x+y)
```

The source code for the above skeleton expression tree calculator program is available.

Exercises

1. Extend the data type `ExprTree` definition and the `eval` function to add the following new kinds of nodes: `Sub`,`Mul`, and `Div` for subtraction, multiplication, and division of values, respectively; `Neg` for negating a value, and `Sin` and `Cos` for the sine and cosine trigonometric functions, respectively.
2. Extend function `derive` to support the operators in the previous exercise.
3. Extend the `simplify` function to support the new operators in the previous exercises. This function should simplify the tree by evaluating all subexpressions involving only constants (not evaluating variables) and handling special values like identity and zero elements.
4. Extend the simplifications in other ways. For example, you could take advantage of mathematical properties such as associativity (`(x + y) + z = x + (y + z)`) and commutativity (`x + 1 = 1 + x`).
5. Write an object-oriented program (e.g. in Java, Scala, or Python

    3) to carry out the same functionality using a class hierarchy and the message-passing style.

Acknowledgements

For the Haskell-based CSci 556 course in Spring 2017, I converted the Expression Tree Calculator case study from Scala to Haskell and adapted this document

from my *Notes on Scala for Java Programmers*, which is itself adapted from the tutorial *Scala for Java Programmers* by Michel Schinz and Phillipp Haller.

Later in Spring 2017, I expanded this case study into an assignment for CSci 556. In 2017 and 2018, I further expanded it into chapters of the textbook now titled *Exploring Languages with Interpreters and Functional Programming*. But, for now, I am keeping this as a separate document.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

References

TODO: Add

Concepts

TODO: Add