The University of Hong Kong

COMP3259: Principles of Programming Languages

Assignment 1

Tutor
Litao Zhou `ltzhou@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

Deadline: 23:59, 26 February 2023

Table of Contents

Note

Your solution must be submitted in a single zip file, named as *A1_XXX.zip*, with *XXX* replaced by your UID. Please submit your solution on Moodle before the deadline. Your code should be well-written (e.g. proper indentation, readable names, and type signatures). You can test your code using `stack test`. Please at least make sure your code compiles!

Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.

1 Extending the Calculator

In the second tutorial, we have learned how to write a simple calculator for arithmetic expressions, including addition, subtraction, multiplication, division, exponentiation, and negation. As a recap, please first answer the following questions:

```
      Div
     /   \
    /     \
   Add    Sub
  / |      | \
 /  |      |  \
5  Power   6   Mul
   / \         /  \
  5   2       3    1
```

Question 1. (5 pts.) Please write down the corresponding Haskell definition for the expression represented by the abstract syntax tree above in `Declare.hs`.

Question 2. (5 pts.) What's the result of evaluating this expression? (Please answer it as comments in `Declare.hs`)

Now let's make our calculator more powerful!

Question 3. (20 pts.) Your first task is to extend the datatype `Exp` with the following operations.

- `Fact` (`x!`): factorial of an integer;
- `Mod` (`x % y`): modulo operation (remainder after division)

We have already added the new operations in `Declare.hs`. Please complete the definition of `evaluate` and `showExp`. For this question, we assume that the `evaluate` function only takes valid `Exp` datatype expressions as input, so you do not need to worry about issues like division by zero for now.

We have provided a parser that supports factorial and modulo operators for you. You can also test your evaluator with the `parseExpr` or `calc` function with a concrete syntax expression as input. The operator `%` has the same precedence as `*` and `/`; and `!` has the same precedence as `^`.

2 Taming Eithers

Question 4. (10 pts.) Complete the definition of `evaluate2` with factorial and modulo operations. The function should report error messages for invalid factorial expressions:

- Evaluating `n!` when `n` is a negative number will report: `Factorial of a negative number: [expression of n]`
- Taking the modulo of zero will report: `Divided by zero: [the expression that produces zero]`

If you are such a programmer who has a good taste of code, you won't be much happy with your solution for `evaluate2` so far. Though it works, the code is ugly, and the nested creeping indentation may drive you crazy.

Notice that there is a pattern recurring in the code. Every time we evaluate some expressions, we get an `Either` value, we do pattern matching on it and fork the computation: if the result is `Right`, we make the contained value available to the rest of computation; if the result is `Left`, we skip the rest of the computation and propagate the error message.

A good programmer is able to recognize the pattern, and try to abstract it to improve the code. Let's do it!

Let's start with a function called `flatMap`, which accepts an `Either` value and a function that encapsulates the rest of the computation, and returns another `Either` value. More specifically, `flatMap` has the following type signature:

```
flatMap :: Either a b -> (b -> Either a b) -> Either a b
```

Don't get intimidated by the type signature above! The definition is just like what we had in the `evaluate2` function: We do pattern matching on the first argument, if it is `Left`, we just return it; If it is `Right`, we apply the second argument (which is a function) to the value contained in `Right`, so that we get another `Either` value as the return value.

Question 5. (10 pts.) Complete the definition of `flatMap`.

Having `flatMap` functions at our service, we can start to refactor `evaluate2` to make it neater.

Question 6. (10 pts.) Create a new function `evaluate3` that is similar to `evaluate2`, but uses `flatMap`. Note that now you don't have to explicitly deal with `Either` values (e.g. pattern matching), except when you have to create them. The resulting code should be significantly simpler than `evaluate2`.

After answering the questions above, you may appreciate the power of abstraction. Actually this kind of pattern has a bizarre name in the functional programming literature: we call `Either` a *monad*. Don't worry if you don't understand it at the moment. We will teach you more about it in the future!

3 Eliminating Zeros

Imagine that we want to evaluate the following expression:

$$(1 + 2 \times 3 + 4 \div 5 - 6 + 7^8 - 9) \times 0$$

We may work very hard in evaluating the left sub-expression inside the parentheses. Once we finish it, we will see that it is immediately multiplied by 0, which gives 0 eventually! In this case, it's endurable; after all, we only did evaluation involving 8 operators and 9 numbers. But how about thousands of operators and numbers? I wish I could know in advance that the result is going to be 0 anyway, so I can save my efforts!

Let's write a function that can help us detect it.

Question 7. (20 pts.) Define a new function `elim0` that does the following on all sub-expressions of an expression:

- Simplify $n + 0$ and $0 + n$ to n;
- Simplify $n - 0$ to n;
- Simplify $n \times 0$ and $0 \times n$ to 0;
- Simplify $0 \div n$ to 0;
- Simplify n^0 to 1;
- Simplify 0^n to 0;
- Simplify -0 to 0;
- Simplify $0 \% n$ to 0;
- Otherwise leave the expression unchanged.

For example:

```
*Interp> elim0 (Mult (Num 3) (Num 0))
0
*Interp> elim0 (Mult (Num 3) (Num 2))
(3 * 2)
*Interp> elim0 (Add (Num 4) (Neg (Mult (Num 3) (Num 0))))
4
```

Note. The `elim0` optimization is not an equivalent optimization. For example, `elim0 (Div (Num 0) (Num 0))` will return `(Num 0)`, which can be safely evaluated to value `0`, while evaluating the expression before the optimization (`evaluate (Div (Num 0) (Num 0))`) will raise an error. Nevertheless, `elim0` is still a sound optimization, because all expressions that can be safely evaluated before the optimization will be evaluated to the same value after the optimization.

4 Inference Rules

In the task 'Eliminating Zeros', we do optimization by transforming the expression into another expression, and then evaluating the new expression. Now, let's do optimization by changing the operational semantics.

Recall that in the lecture, we have shown the inference rule for multiplication and division, namely $E\times$ and $E/$:

$$\frac{e_1 \to n_1 \qquad e_2 \to n_2}{e_1 \times e_2 \to n_1 \times n_2}\ \mathrm{E}\times \qquad\qquad \frac{e_1 \to n_1 \qquad e_2 \to n_2}{e_1/e_2 \to n_1/n_2}\ \mathrm{E}/$$

On top of the existing rules (namely, $E_n, E_+, E_-, E_\times, E_/$) covered in the lecture, we now add three extra rules:

$$\frac{e_1 \to n_1 \qquad n_1 = 0}{e_1 \times e_2 \to 0}\ \mathrm{E}\times\mathrm{L0} \qquad \frac{e_2 \to n_2 \qquad n_2 = 0}{e_1 \times e_2 \to 0}\ \mathrm{E}\times\mathrm{R0} \qquad \frac{e_1 \to n_1 \qquad n_1 = 0}{e_1/e_2 \to 0}\ \mathrm{E}/0$$

Note. For the same reason in the previous task, the new semantics after extension are not equivalent to the original one, but they work for the purpose of optimization.

Question 8. (10 pts.) Write a derivation tree for the expression e below with the new set of inference rules, using the optimized rule.

$$e = ((4+8) \times (1-1))/(16 \times 3)$$

Question 9. (10 pts.) Based on our new inference rules, can there be more than one derivation? If so, show an example of two different derivations for this same expression. If not, explain why not.

The answers to the two questions above can be written in any form (e.g. hand-written, typeset, or ASCII art). Please remember to pack a soft copy of your answers into the zip file you submit.