Notes from 4 Sep

2019-10-21 09:53
e8fd88f

Some FP concepts

- Referential transparency
- Side effects
- Pure functions/languages

Algebraic reasoning about program expressions and equality. These two 'should' be equivalent: $3 * e^2 + 4$ and $3 * e * e + 4$. Consider if e is a C++ function like:

```
int f1(int y) {
  return y+1;
}
```

This function `f1` is pure (no side effects), so the equivalence works:

```
   3 * e^2 + 4 where e = f1(5)
== 3 * f1(5)^2 + 4
== 3 * 6^2 + 4
== 3 * 36 + 4
== 108 + 4
== 112

   3 * e * e + 4 where e = f1(5)
== 3 * f1(5) * f1(5) + 4
== 3 * 6 * f1(5) + 4
== 3 * 6 * 6 + 4
== 108 + 4
== 112
```

But consider instead this function:

```
int f2(int y) {
  cout << "Hello!";
  return y+1;
}
```

```
   3 * e^2 + 4 where e = f2(5)
== 3 * f2(5)^2 + 4
== 3 * 6^2 + 4        // Outputs "Hello!"
== 3 * 36 + 4
== 108 + 4
== 112
```

```
   3 * e * e + 4 where e = f2(5)
== 3 * f2(5) * f2(5) + 4
== 3 * 6 * f2(5) + 4   // Outputs "Hello!"
== 3 * 6 * 6 + 4       // Outputs "Hello!" again
== 108 + 4
== 112
```

The function `f2` does have a side effect: its output. Although both expressions produce the same final value, they can be distinguished by how many times "Hello!" appears.

We can do even worse than this. Here's another type of side effect:

```
int g = 8;  // global variable
int f3(int y) {
  g++;
  return y+g;
}
```

```
   3 * e^2 + 4 where e = f3(5)
== 3 * f3(5)^2 + 4    // g becomes 9
== 3 * 14^2 + 4
== 3 * 196 + 4
== 588 + 4
== 592

// Let's assume g is reset to 8 before proceeding

   3 * e * e + 4 where e = f3(5)
== 3 * f3(5) * f3(5) + 4  // g becomes 9 in first application of f3
== 3 * 14 * f3(5) + 4     // g becomes 10 in next application
== 3 * 14 * 15 + 4
== 630 + 4
== 634
```

Now we get completely different results.

Another typical violation of referential transparency is any function that returns random numbers. This also amounts to modification of global state.

Haskell and GHC

- GHC = Glasgow Haskell Compiler
- REPL = Read, Eval, Print, Loop

For now, we can use `repl.it`[1] – choose "new repl" and then select Haskell. You can

[1] `repl.it/repls`

create an account, or just save/remember the URL to return to the same environment later.

Type this in the editor box, and push the play button:

```
main = putStrLn "Hello, world!"
```

Below are some variations we explored, although I don't want to get **too** deep into all the mechanisms here because we'll explore them later.

You can split the main function onto multiple lines as long as you use indentation to continue them.

```
main =
  putStrLn "Hello, world!"
```

Or:

```
main =
  putStrLn
  "Hello, world!"
```

If you want to **sequence** multiple print statements, the » operator will do that:

```
main =
  putStrLn "Hello, world!" >>
  putStrLn "Goodbye then," >>
  putStrLn "This has been fun."
```

In that example, the line-breaking can be about anywhere, because it's all one big **expression:**

```
main =
  putStrLn "Hello, world!"
  >> putStrLn
  "Goodbye then," >>
  putStrLn
  "This has been fun."
```

Instead of the » to sequence the distinct actions, you can use "do notation", but it will be fussier about line breaking. (You can continue one action onto the next line, but it will need further indentation.)

```
main = do
  putStrLn "Hello, world!"
  putStrLn
    "Goodbye then,"
  putStrLn "This has been fun."
```

We also did this:

```
main = do
  let greeting = "Hello"
  putStrLn "Enter your name: "
  name <- getLine
  putStrLn (greeting ++ ", " ++ name)
```

Comment syntax is double-dash for line comments, and curly-dash for block comments.

```
{- Block comment
   can span multiple lines -}
-- Line comment
```

Operator and function syntax

Caveat about negation operator: in Haskell, the negative sign (subtraction operator) sometimes causes syntactic confusion:

```
λ> 3 * -2

<interactive>:7:1: error:
    Precedence parsing error
        cannot mix ‘*’ [infixl 7] and prefix `-' [infixl 6]
        in the same infix expression
```

The solution to this is just to use parentheses around negation.

```
λ> 3 * (-2)
-6
```

Function application does not require parentheses. Here are some simple examples of defining small functions right in the REPL.

```
λ> areaCircle r = pi * r**2
λ> volumeSphere r = 4/3 * pi * r**3
λ> areaCircle 5
78.53981633974483
λ> volumeSphere 5
523.5987755982989
```

Notice that we don't need parentheses to invoke those functions. You don't need to say `areaCircle(5)`, just `areaCircle 5` is sufficient. If parentheses do appear, they are used to group arguments together. For example,

```
areaCircle 5 + 2
```

would be interpreted as

```
(areaCircle 5) + 2
== 78.53981633974483 + 2
```

```
== 80.53981633974483
```

but if you want to add first, then use explicit parens:

```
areaCircle (5 + 2)
== areaCircle 7
== 153.93804002589985
```

Identifiers

Functions in Haskell can be named either with alphanumeric identifiers (examples: `sqrt`, `div`, `areaCircle`) or symbolic identifiers (examples: `/`, `**`, `<*>`, `>@»`).

By default, alphanumeric identifiers are defined and invoked using *prefix* notation – the function name appears before any of its arguments. In contrast, symbolic identifiers are defined and invoked using *infix* notation – it is assumed that the operation takes two parameters, and the symbol appears *between* them.

For example, the integer division function (`div`) is alphanumeric and the floating-point division function (`/`) is symbolic. They each take two parameters. So the default way to call them is:

```
div 19 2     -- Prefix because alphanumeric
19 / 2       -- Infix because symbolic
```

However, you can override this. We can use alphanumeric function names with infix notation (as long as the function takes exactly two parameters) by surrounding them with "back ticks" – on most keyboards this is on the same key as the tilde (~) character:

```
19 `div` 2   -- Infix notation with alphanumeric
```

We can also use symbolic function names with prefix notation by surrounding the symbol with parentheses:

```
(/) 19 2     -- Prefix notation with symbolic
```

These notation rules also apply to the function definition. Let's make up an operation of two parameters that adds their squares:

This defines it:

```
x >@@< y = x*x + y*y
```

and here's a sample usage:

```
3 >@@< 4
== 25
```