Functional Programming

Paul Mellies

Lesson 2 : Functions and Functional Programming
Functions

Functions in Haskell work exactly as they do in mathematics:

\[
\text{simple } x = x
\]

- **Function name**: In Haskell, every function starts with a lowercase letter.
- **All functions take at least an argument**:
- **The behavior of the function is described here**. In this case, the function returns its argument.
Functions

The `simple` function takes a single argument and returns the argument untouched.

You can test the function in the interactive mode of GHC:

```
ghci> let simple x = x

ghci> simple 2
2

ghci> simple "dog"
"dog"

ghci> simple simple "cat"
"cat"
```
Functional Programming

Three fundamental principles satisfied by Haskell functions:

- a function always takes an argument,
- a function always returns a value,
- anytime a function is called with the same argument, it returns the same result.
Hidden states in function calls

Typically, the following code

```haskell
  tick()
  if (timeToReset){
    reset()
  }
```

could not be written in Haskell, for the following reasons:

- the `tick` function does not take any argument, and thus must be accessing a value in the environment,

- the `reset` function does not return any value, and thus must be changing a value in the environment,

- the two functions thus interact by changing the program’s state and by producing side effects at execution time.
The dangers of side effects

Side effects produce confusing behavior in standard libraries.

```python
myList = [1,2,3]
myList.reverse()
newList = myList.reverse()
```

This code is correct in Ruby, Python and JavaScript. However, the result is very different in each language!

- **Ruby** → [3,2,1]
- **Python** → None
- **JavaScript** → [1,2,3]

**Exercise:** Can you guess what happens in each case?
Variables

The status of variables in Haskell is a subtle and very interesting issue!

See for instance the issue warmly debated at Stack Overflow.

Variables can be declared as **definitions** in this way:

\[ x = 2 \]

or declared as **local variables** at any point of the program, in that way:

\[ \text{let } x = 2 \]
More precisely, two declarations of the same variable \( x \)

\[
\begin{align*}
x &= 2 \\
x &= 3
\end{align*}
\]

induce an error at compile time, while the program

```haskell
main :: IO ()
main = do
  let x = 2
  let x = 3
  print x
```

compiles just fine and prints the value 3 at execution time.
Typically, the following Haskell source file

```haskell
--file : assign.hs

x = 2
x = 3
```

produces the following error message at compile time:

```ghci
ghci> :load assign.hs
[1 of 1] Compiling Main   ( assign.hs, interpreted )
assign.hs:3:1:
  Multiple declarations of `x'
  Declared at: assign.hs:2:1
          assign.hs:2:1
Failed, modules loaded: none.
```
Conditional branching

Consider the \texttt{calcChange} function with two arguments \texttt{owed} = how much is owed \hspace{1cm} \texttt{given} = how much is given

which computes what the cashier should give back to the customer:

\[
\texttt{calcChange} \ \texttt{owed} \ \texttt{given} = \text{if } \texttt{given} - \texttt{owed} > 0 \\
\text{then } \texttt{given} - \texttt{owed} \\
\text{else } 0
\]

Two things are wrong with this function:

(1) even for a tiny function, \textbf{the code is hard to read}
(2) moreover, \textbf{the main computation (subtraction) is repeated!}

Do you see how one could improve the code?
Conditional branching

The original code for the `calcChange` function can be improved as

```
calcChange owed given = if change > 0
   then change
   else 0

where change = given - owed
```

Here, the `where` clause declares and initializes the local variable `change`. In this way, the computation of the value of `change` is shared. As a result, the code is faster and easier to read.

Note also that `where` reverses the usual order of declaring a variable. This is one main benefit of referential transparency in Haskell.
Referential Transparency

An expression is called

**referentially transparent**

when it can be replaced with its corresponding value without changing the program’s behavior.

This requires that the expression *does not produce side effects*.

Referential transparency is important because it enables the programmer and the compiler to **reason about a program behavior** by rewriting.

Useful for proofs of correctness, optimisation, parallelization, etc.
Typically, thanks to referential transparency, the Haskell code

```haskell
calcChange owed given = if change > 0
  then change
  else 0
where change = given - owed
```

behaves in just the same way as the Haskell code

```haskell
calcChange owed given = if given - owed > 0
  then given - owed
  else 0
```

since the second code is obtained by replacing the expression `change` by the expression with same value `given - owed`.