Programming Paradigms

Unit 11 — Functional Programming with Haskell

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

Outline

Background

- Now we'll study a purely functional programming language: Haskell
  - Was developed in 1990 by a committee of experts combining the best features of existing functional programming languages
  - Named after the American mathematician and logician Haskell Curry
- Haskell is a statically and strongly typed, compiled, pure functional programming language
- Not very surprisingly, the centerpiece of Haskell are functions that have input parameters and compute a result

Referential Transparency

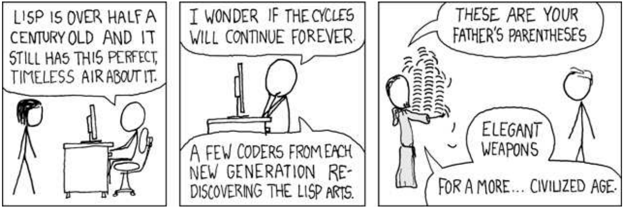
- Referential transparency is a useful property of pure functional languages:
  - functions return the same output, given the same input
  - functions do not have side effects, i.e., they do not modify program state
  - a variable can only be assigned (matched) a value once within a scope or program execution
- Haskell supports referential transparency

Advantages of Referential Transparency

- Allows a compiler to figure out a program's behavior more easily
- Allows a programmer to show correctness of the code more easily
  - Helps in building correct programs by putting together smaller, correct functions, that always behave in the same way
- Allows Haskell to do lazy evaluation: it will not compute anything until the result is actually needed
  - For example, an infinite data structure is not a problem (as long as you don't try to access all of it!)

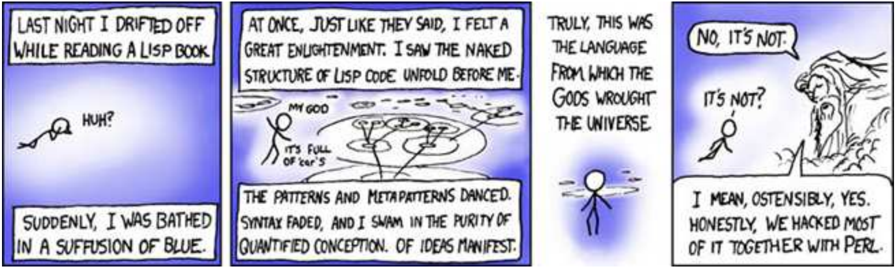
What Do the “Experts” Say?/1

- Functional programming is considered an elegant style of programming



What Do the "Experts" Say?/2

- It is considered to be a bit academic, though



Functional Programming in Practice

- The functional style of programming is applied in practice
- There are users in the financial industry
  - Mainly for building complex models
  - More details are provided here:
    `http://www.haskell.org/haskellwiki/Haskell_in_industry`
- Unreal Engine 4 is a software framework (game engine) designed for the creation and development of video games
  - Has taken functional programming concepts on board, e.g. see here:
    `http://graphics.cs.williams.edu/archive/SweeneyHPG2009/TimHPG2009.pdf`
  - Purists would disagree, as the engine is written in C++, but functional concepts are applied

Starting the Interpreter

- Although Haskell is usually compiled, there is also an interactive interpreter
  - To start the interpreter in Linux, open a console and type `ghci`
- The GHC interpreter prompt > shows up, which means that the interpreter is ready to evaluate an expression
- Here are a few arithmetic expressions

```
> 2+3*4
14
> (2+3)*4
20
> sqrt (3^2 + 4^2)
5.0
```

Hello World

- Let's write a "Hello, world!" program in Haskell

```
Prelude> "Hello, World!"
"Hello, World!"
```

- The Haskell system evaluated the string, and printed the result, which is the string itself.
- We can try a variation to print directly to standard output

```
Prelude> putStrLn "Hello World"
Hello World
```

- Later we will see how to make an exectuable "Hello World" program

Haskell is Strongly Typed

- Haskell is a strongly typed language, it doesn't like you to mix types

```
> 5 + 3
8
> 5 + "string"
<interactive>:8:3:
    ...  some lengthy error message ...
```

- However, in some situations types can be inferred

```
> 2 + 3.5
5.5
```

Variables

- Variables in Haskell begin with a lower-case letter

```
> a = 5
<interactive>:1:3:  parse error on input '='
```

- To assign a value to a variable in the shell, you have to use the function `let`

```
> let a = 5
> a
5
```

- `let` binds the value 5 to the variable a in the local scope (i.e., the console)

Using Functions

- `min` and `max` are two built-in functions with the obvious meaning
- When calling functions, parameters are not enclosed in parentheses, you just list them

```
> min 8 12
8
```

- Functions can be nested inside each other
- Parentheses are used to indicate precedence

```
> max (min 8 12) (min 3 7)
8
```

Writing Your Own Functions/1

- When defining a function of your own in the console, you have to use the function let similar as we did for variable assignments
- The, you have to provide the following parts:
  - The name of the function
  - A list of parameters
  - The symbol =
  - The actual definition (body) of the function

```
> let doubleMe x = x + x

> doubleMe 8
16
```

- The = separates the head of the function from the body of the function, which specifies the actual definition of the function
- The head is also called signature

Writing Your Own Functions/2

- If you want to double two numbers and add them, you could start from scratch

```
let doubleUs x y = x * 2 + y * 2
```

- However, it is good (functional) programming style to re-use correct code

```
let doubleUs x y = (doubleMe x) + (doubleMe y)
```

Conditionals

- Conditionals are functions in Haskell, so they always have to return something:

```
> let doubleSmallNumber x = if x > 100 then x else x*2
```

- Writing statements spanning more than one line in the shell can be a bit of a pain

```
> :{
| let { doubleSmallNumber x = if x > 100
| ;then x
| ;else x*2}
| :}
```

Outline

Lists/1

- Haskell also supports lists with the standard square bracket notation

```
> let numberlist = [1,2,3]
```

- All elements of a list have to be of the same type
- The head and the tail of a list can be obtained by the operator :
- [] represents the empty list

```
> let a:b = numberlist
> a
1
> b
[2,3]
```

- Internally, a list [1] is represented as 1 : []
  - Notice the similarity to Prolog lists that are represented as structures

Lists/2

- You can also extract more than one elements from a list:

```
> let a:b:c = numberlist
> a
1
> b
2
> c
[3]
```

- The `:` operator can also be used to construct new lists

```
> 10:[11,12]
[10,11,12]
```

- Another way is to concatenate two lists with the operator ++

```
> [1] ++ [2,3]
[1,2,3]
```

Lists/3

- Alternatively, you can call the functions head and tail

```
> head numberlist
1
```

```
> tail numberlist
[2,3,4,5]
```

- There are also functions to take and drop the first *n* elements of a list

```
> take 2 numberlist
[1,2]
```

```
> drop 2 numberlist
[3]
```

- There is a large number of other built-in list functions

Lists/4

- You can also create an infinite list!

```
> let naturalNumbers = [1..]
> take 5 naturalNumbers
[1,2,3,4,5]
```

- This works since Haskell is lazy, i.e., Haskell won't execute functions and calculate things until it's really forced to show you a result, e.g., the first five numbers.

Ranges

- Similar to Ruby, you can create lists of numbers in a certain range

```
> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

- You can also skip some numbers or count backwards:

```
> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
```

```
> [10,7..1]
[10,7,4,1]
```

List Comprehensions/1

- Set comprehension is a mathematical way of defining specific sets, given a more general set
- For example, the first ten even natural numbers can be defined by

$$S_{even10} = \{2x | x \in \mathbb{N}, x \leq 10\}$$

- Set comprehensions are usually described by
  - an output function (here $2x$)
  - a variable (here x)
  - an input set (here $\mathbb{N}$)
  - a predicate (here $x \leq 10$)

List Comprehensions/2

- In Haskell this concept can be applied to lists, called list comprehension
- Allows you to generate lists that are too complex for ranges
- For example, out of the first five odd natural numbers, we want those whose square is not equal to 25

```
[ x | x <- [1,3..9], (x*x) /= 25 ]
```

- <- stands for $\in$ (or is interpreted as "drawn from")
- The above list comprehension will output

```
[1,3,7,9]
```

Tuples

- Haskell also knows tuples, which are enclosed in round brackets:

```
(1,"one","uno")
```

- Unlike lists, tuples can combine different data types in the same tuple
- Similar to Prolog structures, except that there is no functor
- Tuples can also be nested

```
(1, ("one", "EN"), ("uno", "IT"))
```

Combining Tuples and Lists

- Consider a triangle in the Euclidean space, which is represented by 3 points; each point is represented by a tuple
- The following list comprehension flips the triangle along the diagonal

```
> [(y,x) | (x,y) <- [(1,2), (2,3), (3,4)]]
[(2,1), (3,2), (4,3)]
```

- This list comprehension has no condition, which means that it is always true
- Shift the triangle horizontally

```
> [(4-x,y) | (x,y) <- [(1,2), (2,3), (3,4)]]
[(3,2), (2,3), (1,1)]
```

Outline

Haskell's Type System/1

- After mentioning types a few times now, it's time to have a closer look
- The `:t` command gives you the type of an expression

```
> :t 'a'
'a' :: Char

> :t True
True :: Bool

> :t "hello!"
"hello!" :: [Char]

> :t (True,'a')
(True,'a') :: (Bool, Char)

> :t 4==5
4==5 :: Bool
```

- All the major built-in types of other languages are also available in Haskell
- Types start with an upper-case letter

Haskell's Type System/2

- You can also find out the type of functions

```
> :t doubleMe
doubleMe ::  Integer -> Integer
```

```
> :t doubleUs
doubleUs ::  Integer -> Integer -> Integer
```

- The last type is the return type
- The others are the type of the input parameters
  - e.g., `doubleUs` has two input parameters of type `Integer` and returns a value of type `Integer`

Type Variables

- Let's look at more subtle typing issues
- For example, what is the type of the function `head`?
- The function `head` can be applied to lists of different types

```
> :t head
head ::  [a] -> a
```

- `a` is a type variable, i.e., `a` can be of any type
- So, the `head` function accepts a list of any type `a` and returns a single element of the same type `a`

Type Classes/1

- In Haskell, types are organized in type classes
- Let's look at the type of the comparison operator?

```
> :t (==)
(==) ::  Eq a => a -> a -> Bool
```

- The symbol => is called a type constraint
  - The left-hand side represents that type variable a has to be a member of type class Eq
  - The right-hand side is the type specification of the function ==
    - two arguments of a type that is a member of the type class Eq and
    - a boolean return type
- Haskell supports a couple of type classes, e.g.,
  - Ord for types that have ordering
  - Num for types that have numerical values

Type Classes/2

- Type classes are similar to interfaces
- They tell you what kind of functions a type supports
- For example,
  - types belonging to the type class `Num` support all the standard mathematical operators: +, -, *, /, ...
  - `Show` converts values to strings
  - `Read` is the opposite: takes a string and converts it to a value

Outline

Writing Modules

- Let's start with some proper programming and define code in a module
- The code below shows a complete module `MyModule`, which we store in a file `MyModule.hs`

```
module MyModule (
doubleMe
) where

doubleMe ::  Integer -> Integer
doubleMe x = x + x
```

- Module names start with an upper-case letter and lists the functions that are exported
- The last two lines are the function definition:
    - the first line specifies the type of the function `doubleMe`,
    - the second line defines the function itself
- Note that the function `let` is not required inside modules

Compiling and Using Modules

- You can load the file `MyModule.hs` into the interpreter with the :l function

```
> :l MyModule
[1 of 1] Compiling MyModule
         ( MyModule.hs, interpreted )
Ok, modules loaded:  MyModule.
*MyModule>
```

- Now you can use the functions defined in the module

```
> doubleMe 2
4
```

- Alternatively, you can also compile the module using the OS command `ghc` and then load the compiled version with `:l` as above

Importing other Modules

- If you want to re-use code from a module in another module, you can import it

```
module YAM (
doubleUs
) where

import MyModule

doubleUs ::  Integer -> Integer -> Integer
doubleUs x y = (doubleMe x) + (doubleMe y)
```