Introduction to Functional Programming

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

Today’s Applications

- Better performance can be obtained via multi-threading
  - Problems:
    - Race Condition
    - Deadlock
  - Need a programming language suitable for multi-threading
- Applications are getting larger and larger
  - Harder to debug
  - Would be better of bugs can be caught during compile time

Programming Paradigms

- **Imperative**: Programmer instructs the machine how to change its state
  - **Procedural**: Instructions are grouped into procedures
  - **Object-Oriented**: Instructions with parts of the state they are operated on are grouped
- **Declarative**: Programmer declares definitions/properties but not how to compute it
  - **Logic**: Desired result is declared as the answer to a question about a system of facts and rules
  - **Mathematical**: Desired result is declared as the solution of an optimization problem
  - **Functional**: Desired result is declared as the value of a series of function application

Programming Paradigms

- Given a list `lst1`, we want to create a new list `lst2` of all elements of `lst1` that are less than 5
- Java (Imparative with Object Oriented)

```
ArrayList<Integer> lst2 = new ArrayList<Integer>();
for(int i = 0; i < lst1.size(); i++) {
    int temp = lst1.get(i);
    if(temp < 5)
        lst2.add(temp);
}
```

- Haskell (Functional)

```
lst2 = [x | x <- lst1, x < 5]
```

Haskell

- **Haskell** is a functional programming language
- Main concept is the **purity** (no side effects)
- Uses **lazy** evaluation
- **Types** are **statically checked** by the compiler
- Has **parameterized polymorphism** and **type classes**

Functional Programming

- Functions are first-class citizens
- Functions can be manipulated like data
- Functions can be passed as arguments
- A return value can be a function
- A function can be assigned to a variable
- **Functions are treated as data**

Function as Data

- Imagine a function named `action` which takes an argument of a type
- To apply this function to every element in a list in Java:

```
Iterator it = listOfThings.iterator();

while(it.hasNext())
{
    Element e = it.next();
    action(e);
}
```

- In Haskell:

```
map action listOfThings
```

  - The function `action` is a parameter to the function `map`
  - The code is also easy to be verified because it is much more concise
  - Java 8 and later embrace functional concepts like the above code

Pure (No Side Effects)

- Consider the following code snippet:

```
int extra;

int addWithExtra(int x)
{
    return x + extra;
}
```

- Calling `addWithExtra(5)` twice may not return the same value
- Other part of the program may modify the value (state) of `extra`
- Elements outside of the program control are called **side effects**:
  - Input and output
  - Network communication
  - Randomness

Pure (No Side Effects)

- A Haskell's code consists of **expressions**
- Expressions are evaluated in the same way as mathematical expression
- Expressions in Haskell cannot have side effects (pure)
- Once a value is assigned to variable, it cannot be changed
- Side effects are possible in Haskell
  - There will be a clear distinction between pure and impure ones
- Improve the ability to reason about the code
  - The outcome of a pure function depends only on its parameters
  - Every run of the same inputs will give the same result
- Order of execution does not matter which is suitable for concurrent execution

Lazy Evaluation

- Consider the function `addPlusOne` as shown below:

```
int addPlusOne(int x)
{
    return x + 1;
}
```

- In Java, if we call `addPlusOne(5 + 9)`, `5 + 9` will be evaluated to 14 first before assigning it to `x`
- In Haskell, `x` will be assigned as the expression `5 + 9`
  - It will not evaluate `5 + 9` until it is necessary to do so
- Lazy evaluation results in a minimal amount of computation
- Need more space in the memory to store expressions that may or may not need to be evaluated

Strong Static Type

- A type system is an abstraction that categorized the values in a program
- Normally used to restrict possible operations on the values
- Types can be checked at the following times:
  - Execution Time (dynamic typing)
  - Compilation Time (static typing)
- Java is a static typed language but needs to perform extra type checking at run-time
- Once a Haskell program is compiled, no more type checks
- A strong typed language has stricter rules at compile time
  - This implies that errors and exceptions are more likely to be caught at compile time

Polymorphism and Type Classes

- Polymorphism:
  - The `ArrayList` in Java can be used to store any types
  - This is a benefit of the **Java Generics** (Template in C++)
  - Haskell supports **parametric polymorphism**
  - A function can be expressed without caring about the type of its parameter(s)
- Type Classes:
  - An interface in Java allows us to group unrelated classes without forcing them to share a common hierarchy
  - A class that implements an interface must implements all methods defined in the interface
  - Type classes in Haskell group different types with a common interface

Installing Haskell

- The current version of Glassgow Haskell Compiler (GHC) can be found at

  ▸ https://www.haskell.org/ghc/download_ghc_9_2_1.html

- For **Windows**, either
  - Download the `.tar.xz` and extract it using either `WinZip` or `7-Zip` or
  - Install one of the UNIX flavors on Windows (does not have to be a virtual machine) and install the `haskell-platform` package
- For **macOS**, I need help on this
- For any **UNIX** flavors such as Ubuntu, simply install the `haskell-platform` package

Simple Test

- Execute the command `ghci`
- `Prelude>` is the default prompt

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> 5 * 3
15
Prelude> 1/2 + 1/3
0.8333333333333333
Prelude> sqrt 7
2.6457513110645907
Prelude> :show language
base language is: Haskell2010
with the following modifiers:
  -XNoDatatypeContexts
  -XNondecreasingIndentation
Prelude> "Hello World!!!"
"Hello World!!!"
Prelude> :quit
Leaving GHCi.
```