20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by Sri Sai Tanvi Sonti
CB.EN.U4CYS21072
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

AMRITA
VISHWA VIDYAPEETHAM

Outline

What is functional paradigm?

- Functional programming is a declarative programming paradigm in which programs are constructed by applying and composing functions.
- Functional paradigm main focus is on **"what to solve"** in contrast to an imperative style where the main focus is "how to solve".
- It is based on the principles of mathematical functions, emphasizing immutability, pure functions, and declarative programming.
- This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

History

The functional programming paradigm has its roots in **lambda calculus**, a mathematical system developed by **Alonzo Church** in the 1930s. Lambda calculus provided a formal foundation for expressing computation through functions and played a crucial role in the development of functional programming.

Key concepts of Functional Programming

- **Immutability**: Data is treated as immutable, meaning that once a value is assigned, it does not change. Instead of modifying existing data, functional programming emphasizes creating new data structures with modified values.
- **First-Class Functions**: Functions are treated as first-class citizens, allowing them to be assigned to variables, passed as arguments to other functions, and returned as values from other functions. This enables the use of higher-order functions.
- **Pure Functions**: A pure function is a function that, given the same input, will always return the same output and has no side effects. Side effects refer to changes in state outside the function, such as modifying global variables or performing I/O operations.
- **Recursion**: Functional programming often relies on recursion for iteration rather than traditional loop constructs.
- **Declarative Style**: Emphasizes expressing what should be done rather than how it should be done. This is in contrast to imperative programming, which focuses on describing how a task should be accomplished.

Real World applications of Functional Programming

- **Finance and Banking**: It helps model complex financial systems, perform risk analysis, and handle large datasets efficiently.
- **Web Development**: JavaScript (with libraries like React and Redux), are commonly used for building modern web applications. Functional concepts, such as pure functions and immutability, contribute to predictable and maintainable code.
- **Telecommunications**: Erlang, a functional programming language, is known for its use in the telecommunications industry. It is designed for building scalable and fault-tolerant distributed systems, making it suitable for telecom infrastructure.
- **Cloud Computing** : Functional programming is applied in cloud computing platforms and services for tasks such as distributed computing, resource management, and handling concurrent requests efficiently.
- **Concurrency and Parallelism**: Languages like Clojure and Haskell support concurrent programming models, making them suitable for applications that require efficient use of multicore processors. accomplished.

Comparison

- **Functional Programming**: Emphasizes the use of mathematical functions to express computations.
- **Imperative Programming**: Describes how a program should achieve a particular result through a sequence of statements.
- **Logical Programming**: Uses formal logic to express computations and derive conclusions.
- **Object-Oriented Programming (OOP)**: Organizes code around objects, which encapsulate data and behavior.

Disadvantages

- Sometimes writing pure functions can reduce the readability of code.
- Writing programs in recursive style instead of using loops can be bit intimidating.
- Writing pure functions are easy but combining them with the rest of the application and I/O operations is a difficult task.
- Immutable values and recursion can lead to decrease in performance.

Scheme

;; Define a function to calculate the factorial of a number (define (factorial n) (if (= n 0) ; Base case: factorial of 0 is 1 1 (* n (factorial (- n 1))))) ; Recursive case: n! = n * (n-1)!
;; Test the factorial function (display "Factorial of 0: ") (display (factorial 0)) (newline) (display "Factorial of 5: ") (display (factorial 5)) (newline) (display "Factorial of 10: ") (display (factorial 10)) (newline)

Explanation

Explanation:

The factorial function is defined using the define keyword.

It uses recursion to calculate the factorial. The base case is when n is 0, in which case the factorial is 1.

The recursive case multiplies n by the factorial of (n-1).

The display and newline functions are used to print the results.

What is concurrent paradigm?

- The concurrent programming paradigm involves the simultaneous execution of multiple tasks or processes, allowing different parts of a program to run independently and potentially in parallel.
- A concurrent system is one where a computation can advance without waiting for all other computations to complete.
- Although both concurrent and parallel computing can be described as **"multiple processes executing during the same period of time"**.
- In parallel computing, execution occurs at the same physical instant (on processors of multi-processor machine).
- By contrast, concurrent computing consists of process lifetimes overlapping, but execution need not happen at the same instant.

History

The academic study of concurrent algorithms started in the 1960s, with Dijkstra (1965) credited with being the first paper in this field, identifying and solving mutual exclusion. Concurrent programming paradigm is intertwined with the evolution of computing hardware, the growing need for efficient resource utilization, and advancements in programming languages and models.

Key concepts of Concurrent Programming

- **Processes**: A process is an independent program that runs in its own memory space and executes independently of other processes.
- **Threads**: Threads enable concurrent execution within a process, allowing for more efficient utilization of resources. Thread synchronization is crucial to avoid conflicts.
- **Shared Memory**: Shared memory allows tasks to communicate by reading and writing to shared data. Synchronization mechanisms (e.g., locks) are required to ensure data consistency.
- **Synchronization**: Synchronization mechanisms include locks, semaphores, and barriers. They prevent race conditions, deadlocks, and ensure data consistency.

Real World applications of Concurrent Programming

- **Operating Systems**: Concurrent programming is fundamental to the design of modern operating systems.
- **Web Servers**: Web servers handle multiple incoming requests concurrently. Each incoming request is processed independently, allowing the server to serve multiple clients simultaneously and maintain responsiveness.
- **Internet of Things (IoT) Devices**:IoT devices often require concurrent programming to handle multiple sensor inputs, communicate with other devices, and perform real-time data processing.
- **Game Development** : Concurrent programming is used in game development for tasks such as rendering graphics, handling user input, and managing AI.

Disadvantages

- **Race Conditions**: ace conditions occur when the behavior of a program depends on the relative timing of events in different threads or processes.
- **Deadlocks**: Deadlocks can occur when two or more tasks are unable to proceed because each is waiting for the other to release a lock. Detecting and resolving deadlocks can be challenging and may require careful design and analysis.
- **Complexity**:Concurrent programming introduces complexity due to the need for synchronization, communication, and coordination between concurrent tasks. Dealing with shared resources, locks, and avoiding race conditions can make code more intricate and error-prone.

```
-module(concurrentexample).
-export([start/0, worker/1, supervisor/2]).
worker(List) ->
Sum = lists:sum(List),
io:format("Worker  w calculated sum:  w n", [self(), Sum]),
Sum.

supervisor(Worker1, Worker2) ->
receive
Worker1, Sum1 ->
receive
Worker2, Sum2 ->
TotalSum = Sum1 + Sum2,
io:format("Supervisor received sums:  w and  w, Total Sum:
w n", [Sum1, Sum2, TotalSum]), supervisor(Worker1, Worker2)
end end
start() -> Worker1 = spawn(fun() -> worker([1, 2, 3, 4, 5]) end), Worker2 =
spawn(fun() -> worker([6, 7, 8, 9, 10]) end),
Supervisor = spawn(fun() -> supervisor(Worker1, Worker2) end),
Worker1 ! self(), Worker1 ! [1, 2, 3, 4, 5], Worker2 ! self(), Worker2 ! [6, 7, 8, 9, 10],
timer:sleep(1000),
```

Explanation

Explanation:

The worker/1 function represents a worker process that calculates the sum of a list of numbers and sends the result to its supervisor.

The supervisor/2 function represents a supervisor process that waits for results from both worker processes, calculates the total sum, and performs further processing if needed.

The start/0 function creates two worker processes and a supervisor process. It then sends messages to the workers to start their calculations and waits for the supervisor to receive and process the results.

Benchmarking

- In a functional programming paradigm, there is a strong emphasis on pure functions. Pure functions are functions that, given the same input, always produce the same output and have no side effects. This makes the code predictable and easier to reason about.
- Erlang follows the functional programming paradigm by encouraging the use of immutable data and pure functions. Functions in Erlang do not have side effects by default.
- In functional programming, data is typically immutable. Once a value is assigned, it cannot be changed. Instead, new values are created through transformations.
- Erlang promotes the use of immutable data structures. This is important for ensuring the consistency of data in a concurrent environment.
- Functional programming languages may or may not have built-in support for concurrent programming. Concurrency is often achieved through parallelism, where multiple computations are executed simultaneously.
- Erlang is designed for concurrent programming. It provides lightweight processes (not operating system processes, but rather lightweight threads of execution) that communicate with each other through message passing. Concurrency is a fundamental part of the language, allowing the development of highly concurrent and distributed systems.

References

- https://en.wikipedia.org/wiki/Erlang(programminglanguage)
- https://www.tutorialspoint.com/erlang/erlangbasicsyntax.htm
- http://people.eecs.berkeley.edu/bh/ssch19/implementhof.html
- https://search.brave.com/search?q=example
- https://en.wikipedia.org/wiki/Scheme(programminglanguage)
- https://www.shido.info/lisp/idxscme.html
- https://cs.nyu.edu/ wies/teaching/ppc-14/material/lecture10.pdf
- https://eecs390.github.io/notes/functional.html
- https://www.scheme.org/
- https://www.composingprograms.com/pages/32-functional-programming.html
- https://medium.com/@staneyjoseph.in/understanding-the-basics-of-scheme-for-functional-programming-with-ai-features-1e2a32e7f3f1
- https://www.erlang.org/doc/gettingstarted/concprog.html
- https://uniwebsidad.com/libros/concurrent-erlang/chapter-1/concurrency
- https://gorillalogic.com/blog/playing-with-erlang-concurrency
- https://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency