Faculty of Informatics and Computer Science

Software Construction & Testing

Topic 4

Functional Programming II

(Exploring Advanced Concepts in Functional Programming)

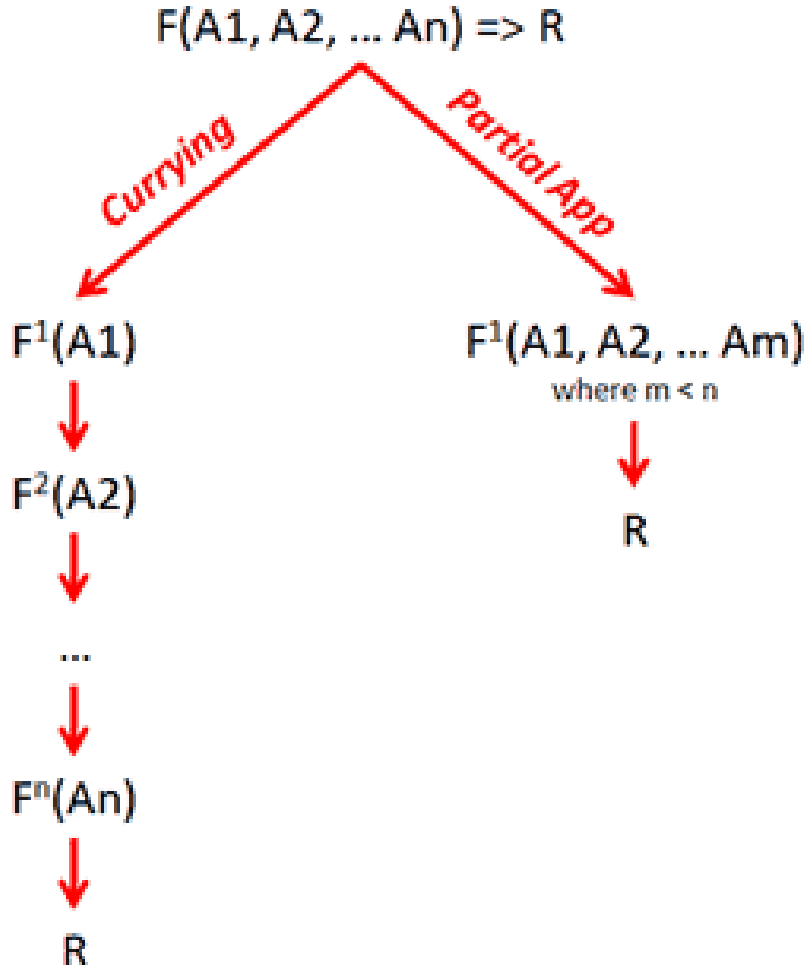
Dr. Ahmed Maghawry

Contents

1. Currying & Partial Application

Currying and partial application are techniques used to transform functions with multiple arguments into functions with fewer arguments or a sequence of functions with a single argument.

Both techniques are based on the concept of higher-order functions, which treat functions as first-class citizens.



1. Currying & Partial Application

Currying is the process of converting a function that takes multiple arguments into a sequence of functions, each taking a single argument.

The curried function takes the first argument and returns a new function that takes the second argument, and so on, until all arguments are consumed and the final result is returned.

```
function add(x) {
  return function(y) {
    return x + y;
  }
}

const add5 = add(5); // The first argument is 5
console.log(add5(3)); // Outputs 8
```

1. Currying & Partial Application

Partial application is closely related to currying and refers to the process of fixing a function's arguments to create a new function with fewer arguments.

It allows you to create specialized versions of a function by providing some, but not all, of its arguments. The resulting function can then be called with the remaining arguments at a later time.

```
const add2 = add(2); // The first argument is 2
console.log(add2(3)); // Outputs 5
```

2. Type Systems & Type Inference

Type Systems

A type system is a set of rules and constraints that assign types to variables, expressions, and functions in a program.

It ensures type safety by enforcing strict adherence to type rules and preventing incompatible operations.

```
int x = 10;
string message = "Hello";
bool flag = true;

int result = x + message; // Error: Incompatible types
```

Type Inference

It is a feature that allows the compiler to automatically deduce the types of variables based on their usage and the context in which they are declared.

It eliminates the need for explicit type annotations in certain cases, making the code more concise and readable.

```
var age = 25;
var name = "John";
var isStudent = true;

Console.WriteLine(age.GetType());      // Output: System.Int32
Console.WriteLine(name.GetType());     // Output: System.String
Console.WriteLine(isStudent.GetType()); // Output: System.Boolean
```

3. Laziness & Infinite Data Structures

Laziness

It refers to a programming technique where expressions or computations are deferred until their values are actually needed.

This allows for on-demand evaluation and can provide performance benefits by avoiding unnecessary computations.

```
Lazy<int> lazyValue = new Lazy<int>(() => ComputeExpensiveValue());

int result = lazyValue.Value; // The value is computed when it is f
irst accessed
```

Infinite Data Structures

These are data structures that can represent an unbounded or potentially infinite amount of data.

These structures enable working with sequences of data without explicitly defining their size or generating all the elements upfront.

```
public static IEnumerable<int> InfiniteNumbers()
{
    int i = 0;
    while (true)
    {
        yield return i++;
    }
}

// Usage:
var numbers = InfiniteNumbers().Take(5);
foreach (var number in numbers)
{
    Console.WriteLine(number);
}
```

4. Pattern Matching

Pattern matching

Is a technique used to match the structure of data against predefined patterns and perform different computations or actions based on the match.

```
public static string DescribeNumber(int value)
{
    return value switch
    {
        0 => "Zero",
        1 => "One",
        2 or 3 => "Two or Three",
        var n when n > 3 => "Greater than Three",
        _ => "Unknown"
    };
}

string numberDescription = DescribeNumber(5);
Console.WriteLine(numberDescription);
```

5. Concurrency & Parallelism

Concurrency

allows for the execution of multiple tasks or computations independently, potentially overlapping in time. It enables effective management of different tasks, maximizing resource utilization and responsiveness.

Concurrency is useful when tasks can progress concurrently, even if not executed simultaneously.

```
public async Task<string> GreetAsync(string name)
{
    await Task.Delay(1000); // Simulate an asynchronous operation
    return $"Hello, {name}!";
}

// Concurrently greet multiple people
var tasks = new List<Task<string>>();
tasks.Add(GreetAsync("Alice"));
tasks.Add(GreetAsync("Bob"));
tasks.Add(GreetAsync("Charlie"));

await Task.WhenAll(tasks);
```

Parallelism

Involves the simultaneous execution of multiple tasks or computations. It aims to improve performance by dividing a larger task into smaller subtasks that can be executed concurrently.

Parallelism is beneficial for computationally intensive tasks that can be divided into independent units of work.

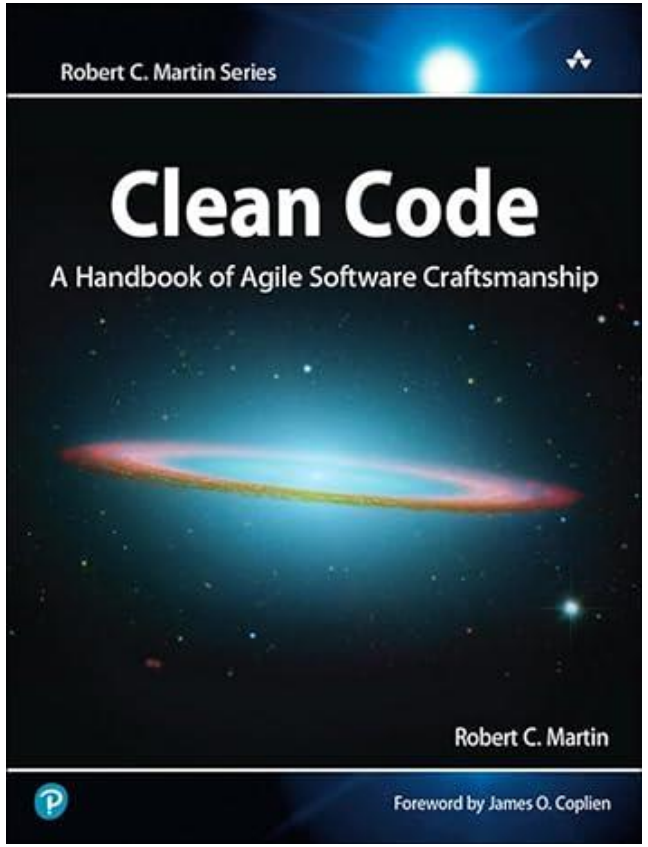
```
var numbers = Enumerable.Range(1, 1000);
var result = 0;

Parallel.ForEach(numbers, number =>
{
    result += Compute(number); // Perform a computation on each num
ber
});
```

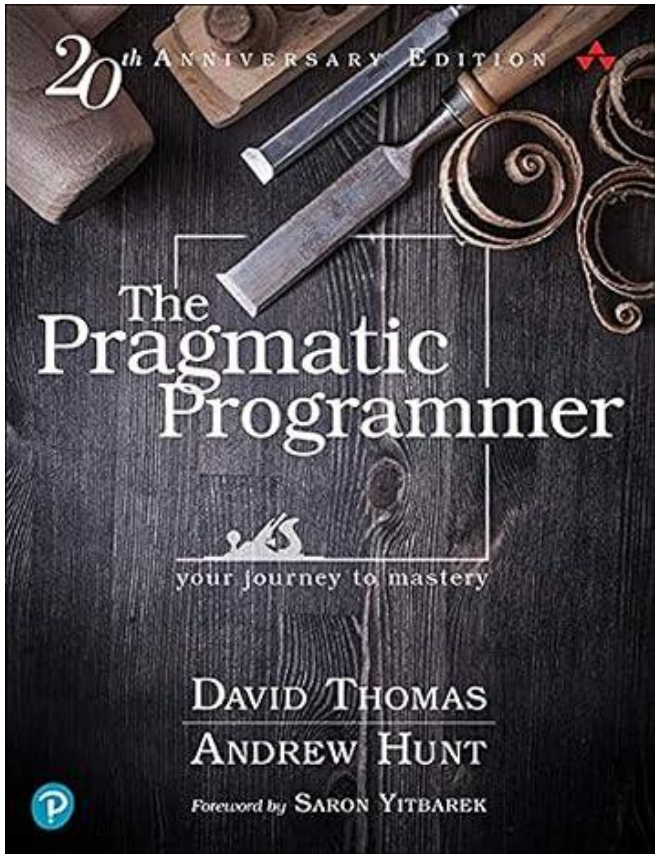
6. Conclusion

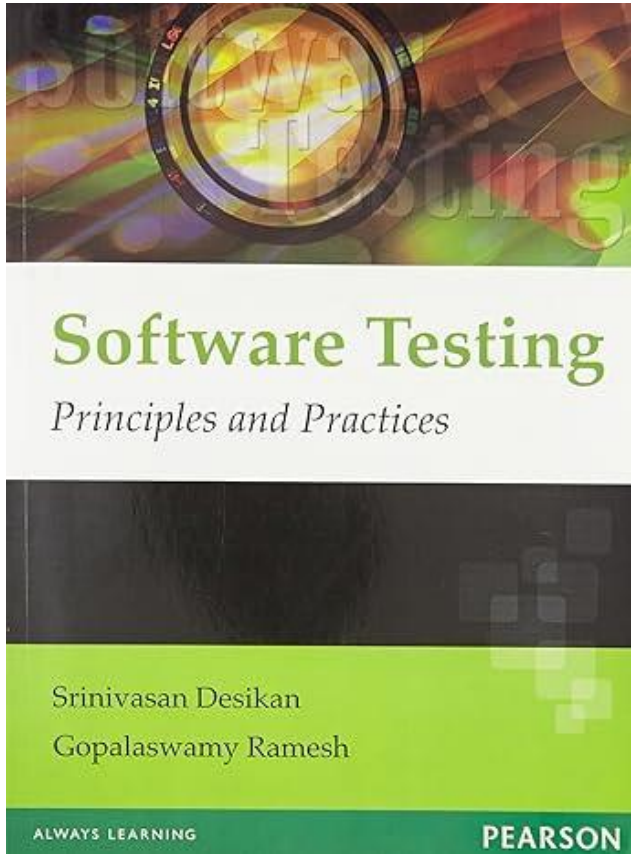
- **Currying and Partial Application:**
  - Promotes code modularity and reusability by transforming functions into sequences of specialized functions or fixing arguments. They enable composability and flexibility in functional programming.
- **Type Systems and Type Inference:**
  - Ensures program correctness and safety by providing static type checking. They reduce the need for explicit type annotations and facilitate robust software development.
- **Laziness and Infinite Data Structures:**
  - Laziness allows for delayed computations and manipulation of potentially infinite data structures. It improves efficiency and enables concise representation of large or infinite datasets.
- **Pattern Matching:**
  - Simplifies complex data structure handling through concise and structured actions based on matching patterns. It enhances code readability and logic in functional programming.
- **Concurrency and Parallelism:**
  - Allows for independent execution of multiple tasks, while parallelism achieves simultaneous execution to improve performance. These techniques enhance responsiveness, scalability, and efficiency in functional programs.

References

- Martin, R. C. Clean code: A handbook of agile software craftsmanship. Prentice Hall.
- Hunt, A., & Thomas, D. The pragmatic programmer: Your journey to mastery. Addison-Wesley.
- Desikan, S., & Ramesh, G. Software testing: Concepts and practices. Pearson Education.







Faculty of Informatics and Computer Science

Questions?