Comparative Analysis of Multi Paradigms Languages

Shad Ahmed
i14-1028
MS(CS)

Faisal Mumtaz
i16-1024
MS(CS)

Umar Munir
i16-1011
MS(CS)

Sharan Gohar
i16-1064
MS(CS)

Mehreen Alam
i16-1402
Phd(CS)

Shahid Hussain
i17-1053
MS(CS)

ABSTRACT

Multiprogramming paradigms has gained immense popularity because of the wide spectrum of different paradigms covered. Not much work has been done to perform any comparative study and/or analysis of how various features are mapped onto different programming languages. This paper aims to highlights the concepts and usage of various features of different programming languages in multiprogramming languages. We have shortlisted our work to twelve programming languages Scala, Swift, Falcon, F#, Rust, VB.net, C#, Oz, Mozart, Matlab, R, and Python. On the other dimension, the features we have chosen to enhance our understanding of multiprogramming paradigm are Bound Checking, Type Safety, Exception Handling, Modularity, Compiled/Interpreted, Assertion, File Handling, Mutable, Immutable, Imperative Control and Explicit Concurrency. To the best of our knowledge, this is the first attempt to explore different features in depth and correlate them in their perspective of the above mentioned programming languages. The study revealed interesting patterns. While all languages support type safety, assertion, file-handling, exception handling, there is a variation in the behavior of the multi-paradigm programming languages when it comes to bound checking, meta-programming, compiled/interpreted, immutability, imperative control and explicit concurrency.

Keywords

theory of programming languages, multi-programming paradigms, features of programming languages

1. INTRODUCTION

Programming paradigms are a way to classify programming languages based on their features. Languages can be classified into multiple paradigms. Some paradigms are concerned mainly with implications for the execution model of the language, such as allowing side effects, or whether the sequence of operations is defined by the execution model. Other paradigms are concerned mainly with the way that code is organized, such as grouping a code into units along with the state that is modified by the code. Yet others are concerned mainly with the style of syntax and grammar. Common programming paradigms include imperative, declarative and symbolic. The former allows side effects with object-oriented which groups code together with the state the code modifies and procedural which groups code into functions. Declarative which does not state the order in which operations execute with functional which disallows side effects and logic which has a particular style of execution model coupled to a particular style of syntax and grammar. The latter has a particular style of syntax and grammar as mentioned by [18, 10].

For example, languages that fall into the imperative paradigm have two main features: they state the order in which operations occur, with constructs that explicitly control that order, and they allow side effects, in which state can be modified at one point in time, within one unit of code, and then later read at a different point in time inside a different unit of code. The communication between the units of code is not explicit. Meanwhile, in object-oriented programming, code is organized into objects that contain state that is only modified by the code that is part of the object. Most object-oriented languages are also imperative languages. In contrast, languages that fit the declarative paradigm do not state the order in which to execute operations. Instead, they supply a number of operations that are available in the system, along with the conditions under which each is allowed to execute. The implementation of the language's execution model tracks which operations are free to execute and chooses the order on its own.

Our work is a detailed study but not an exhaustive one. We have shortlisted the following multi-programming paradigm languages: Scala, Swift, Falcon, F#, Rust, VB.net, C#, Oz, Mozart, Matlab, R, and Python. Out of the comprehensive feature list we have focused our scope to the following only: Bound Checking, Type Safety, Exception Handling, Modularity, Compiled/Interpreted, Assertion, File Handling, Mutable, Immutable, Imperative Control and Explicit Concurrency. To the best of our knowledge, this is the first attempt to explore different features in depth and correlate them in their perspective of the above mentioned programming languages. The study revealed interesting patterns. While all languages support type safety, assertion, file-handling, exception handling, there is a variation in the behavior of the multi-paradigm programming languages when it comes to bound checking, meta-programming, compiled/interpreted, immutability, imperative control and explicit concurrency.

WOODSTOCK '97 El Paso, Texas USA

The paper is organized as follows. Section I introduces the concept of multi-programming paradigm and its various features while section 2 explains the literature review. All the features are discussed in detail in section 3 followed by detailed discussion analysis in section 4. We finally conclude in section 5 and mention the future work in section 6.

2. RELATED WORK

Most Programming languages provide support for basic generic programming. Generic programming aims to express algorithms and data structures in a way to be broadly adaptable [22]. Declarative programming languages provide more abstract level programming than imperative languages [13]. They are characterized by their formal nature. Modern programming languages are not just object-oriented but include multiple paradigms. Common programming paradigms are imperative, declarative and symbolic [19]. Use of multiple programming paradigms help programmers in choosing suitable programming styles to get the job done [4]. Existing work demonstrated that separate paradigms for solving problems is insufficient[3]. Hence analyzing multi-paradigm languages should be under study in order to manage work from multiple perspective.

3. FEATURES

3.1 Bound Checking

In computer programming, bound checking is any method of whether variable detecting variable is within bound before it is used. A failed bounds check usually results in the generation of some sort of exception signal.

3.1.1 Range checking

It is usually used to check that whether a number fits into a given type. A range check is a check to make sure a number is within a certain range; for example, range check will ensure that a value that will assign to a 16-bit integer is within the capacity of a 16-bit integer. Some range checks may be more restrictive; for example, a variable to hold the number of a calendar month may be declared to accept only the range 1 to 12.

3.1.2 Index checking

In index checking, a variable being used as an array index is within the bounds of the array. Index checking means all expressions indexing an array, the index value is checked against the bounds of the array, which were created when the array was defined, and if the index is out-of-bounds, an error occur and further execution is suspended. If a number outside of the upper range is used in an array, it may cause the program to crash, or may introduce security vulnerabilities, index checking is a part of many high-level languages.

3.1.3 Examples

3.1.4 Scala

Array representation in scala
scala> val a1 = Array(1, 2, 3)
Array[Int] = Array(1, 2, 3)

Table 1: Index Checking

	Index checking	Range checking
Scala	✔	✔ (statically check)
Swift	✔	✔
F#	✔	✔
Rust	✔ (at run time)	-
Vb.net	✔	✔
C#	✔	✔
D	✖	✔
Oz	-	-
Matlab	✔	✔(statically check)
Python	✔	✔

3.1.5 Swift

func contains(Bound) Returns a Boolean value indicating whether the given element is contained within the range.

3.2 Type Safety

The compiler will validate types and through an error if you assign a wrong type to a variable. Type safety is checking for matched data types during compile time. For example, int a ="John" returns error as variable 'a' is an integer and we are assigning a string value. These data type mismatches are checked during compile time. Type safe code can access only the memory locations that it has permission to execute. Type safe code can never access any private members of an object. Type safe code ensures that objects are isolated from each other and are therefore safe for inadvertent or malicious corruption At compile time, we get an error when a type instance is being assigned to an incompatible type; hence preventing an error at run-time. So at compilation time itself, developers come to know such errors and code will be modified to correct the mistake. So developers get more confidence in their code. Run time type safety ensures, we don't get strange memory exceptions and inconsistent behavior in the application.

3.2.1 Scala

Scala is strongly type and smart about static type. Scala has powerful type inference. It will figure out itself mostly no need to tell it the types of your variables.

3.2.2 Swift

Swift is type safe as it performs type checks when compiling code and flags any mismatched types as errors. This help in early catch and fix error in the development process. It provides type inference which basically means that coders do not require to spend more time in defining what types of variables they are using.

3.2.3 F#

In F#, static type checking can be used as an instant unit test, making sure that your code is correct at compile time. F# is more type-safe than C# since F# compiler can catch errors that would only be detected at run-time in C#.

3.2.4 Rust

Rust is a type-safe language. Rust has an escape valve

from the safety rules when there is a need to use a raw pointer. This is called unsafe code, and while most Rust programs do not need it, how to use it and how it fits into Rustś overall safety scheme in [3]
https://www.safaribooksonline.com/library/view/programming-rust/9781491927274/ch21.html#unsafe-code

3.2.5 VB.net

Type safety in .NET has been introduced to prevent the objects of one type from peeking into the memory assigned for the other object.

3.2.6 C#

Type safety prevents assigning a type to another type when are not compatible.

public class Employee public class Student

In the above example, Employee and Student are two incompatible types. We cannot assign an object of employee class to Student class variable. An error occurs during the compilation process if any such attempt is done. Type safety check happening at compile time are called static type checking Cannot implicitly convert type 'Program.Employee' to 'Program.Student'.
When tried to type cast object of wrong type. We get
Unable to cast object of type 'first object' to 'second object' type checking happens at runtime, hence it is called runtime type checking

3.2.7 D

D has compile-time type safety.

3.2.8 OZ

OZ also known as MOZART. Oz variables are single-assignment variables or more appropriately logic variables. A single assignment variable has a number of phases in its lifetime. Initially, it is introduced with unknown value while later it might be assigned a value and we say variable has become bound. Once a variable is bound, it cannot itself be changed.

3.2.9 Matlab

MATLAB is a loosely or weakly-typed language. Difference between MATLAB and a strongly-typed language is that there is no need to explicitly declare the types of the variables to use. For example, the declarations x=5; x='foo' immediately following one another are perfectly acceptable; the first declaration causes x to be treated as a number, the second changes its treatment to a string

3.2.10 Python

Python or Ruby are often referred to as dynamically typed languages, which throw exceptions to signal type errors occurring during execution

3.3 Exception Handling

An exception handler is a block of code that is executed if an exception occurs during the execution of some other block of code. In this sense, exceptions are a kind of control statement. Raising an exception transfers the flow-of-control to exception handling code. Users can also throw a self created exception.

Table 2: Type Safety

Languages	Type Safety
Scala	Strongly type, Static type, powerful type Inference
Swift	Type check at compile time, Support type inference
F#	Static Type Checking, Compile Time
Rust	Type Safe, escape valve, unsafe to use raw pointers
VB.net	Type safety use for memory security
C#	Static type checking, type checking compile time
D	Type safe, compile time
Oz	Single Assignment variables, Once value is assigned to variable it can never be change
Matlab	Weakly type language, no need to assign type explicitly,
Python	Dynamically type language,

3.4 Meta-programming

Meta-programming is the capability to adapt itself whereas meta stack overflow is the place to ask question about stack overflow itself. We can also say the ability to treat programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running. Meta-programming is not one specific technique, but rather an ensemble of concepts and techniques. There are two different ways of doing meta-programming: on the Syntax level and at Run-time as explained below.

- **Features for Syntax:** These are feature of languages through which Syntax meta-programming apply.
- **Features for Runtime:** These are feature of languages through which Run-time meta-programming apply.

Reflection: is the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime.

3.5 Compiled / interpreted

Compiled/interpreted: The difference lies not in the language but how the language has been implemented. A compiler translates the source code of the program into another language format that can be directly executed by a lower-level machine. This can be an abstract machine (such as .NET or the Java Virtual Machine) or the actual machine. In the latter case, the language format that is the target of the compiler is machine code. The translation from source code into lower-level code depends on the abstract syntax and on the operational semantics of the programming language. An interpreter executes the source code directly; informally, it may help to think of the interpreter as executing the program line by line. A more correct understanding is that the interpreter walks through the abstract syntax tree

Table 3: Exception handling table

Lang.	Throw	Handler	Assertion
Scala	throw	try { instructions } catch (exception) { instructions} finally{instructions}	Assert(statement)
Swift	throw exception ()	do { try expression ... instructions } catch exception { instructions }	assert(condition,description)
Falcon	raise exception	falcon.HTTPError (status,title=None, description= None)	
F#	raise exception	try expression with pattern or try expression finally expression	assert condition
Rust	Err(exception)	match fun_nam(x,y) { Ok(v) =>{ println!("{}", v); }, Err(err) =>{ println!("{}", err); }}	
Vb .Net	throw exception	Try instructions Catch exception When condition instructions ... Finally instructions End Try	Debug.Assert(condition)
C#	throw exception;	try { instructions } catch (exception) { instructions} finally {instructions}	Debug.Assert(condition);
D	throw	try { instructions } catch (exception) { instructions} finally {instructions}	
Oz	{exception. 'raise'X}	try S catch Pattern_1 then S1 Pattern_2 then S2 finally S_final end	
R	throw()	tryCatch({ expr}, war=function(w){ warning-handler-code}, error = function(e) {error-handler-code}, finally = { cleanup-code }	assertError (expr,verbose=FALSE)
Python	raise exception	try: Tab instructions except exception: Tab instructions else: Tab instructions finally: Tab instruction	assert condition

Table 4: Meta Programming table

Prog. Lang.	Ways of Doing		Features for compile time	Features for Run Time
	Compile Time	Run Time		
R		✓		Objects
Scala	✓	✓	Reflection	Macros
Swift	✓			Templates
Falcon	✓			Macro
F#	✓	✓		Quotation
Rust	✓			Macros
VB.net	✓	✓	Reflection	Reflection
C#		✓		Objects
D	✓			Template
Oz	x	x		
Matlab	x	x		
Python		✓		meta-classes

generated by the parser and executes each node in this tree. If a node is a leaf, the leaf is executed. If a node is an internal node, each sub-tree is visited and executed. Exactly how this is to be done depends on the abstract syntax and on the underlying semantics of the programming language. Out of the many programming languages in this world, some of them are called compiled languages while some are interpreted. For compilation, the software uses is called compiler while for interpreter is used for interpreted language. For a compiled language, an interpreter can be built but the reverse is impossible. That is, all the interpreted languages cannot be a compiled language. Additionally, being interpreted or compiled is not the property of the programming languages, but the design of some languages make them unsuitable for native code generation.

3.6 Assertion

Assertion: specifies that a program satisfies certain conditions at particular points in its execution. An assertion violation indicates a bug in the program. Thus, assertions are an effective means of improving the reliability of programs and function as a systematic debugging tool. There are three types of assertion: pre-conditions, post-conditions and invariants. Preconditions specify conditions at the start of a function; post-conditions specify conditions at the end of a function while invariants specify conditions over a defined region of a program. Asserts are to be used primarily for checking parameter types, classes, or values, checking data structure invariants, checking "can't happen" situations (duplicates in a list, contradictory state variables) or after calling a function, to make sure that its return is reasonable. However, asserts are not to be used for handling runtime errors, like entering a negative number when positive is needed. But used to catching the program errors.

3.7 Conditional compilation

Conditional compilation is a method of producing different results by different parameters provided during compilation of the program. This technique is used when a program is built for different platforms or to run or not to run specific

Table 5: Compiled/interpreted]

	compiled	**interpreted**
Scala	actually a compiled , wherein everything you type gets compiled to the byte code and it runs within the JVM.	illusion of interpreted
Swift	compiled	
Falcon	compiled, The Falcon compiler contains a meta-compiler[23] that supports macro expansions. A Falcon Virtual Machine in the standard compiler drives the meta-compiler. Output generated from the meta-compiler is sent to the language lexer as if part of the original source.	
F#	compiled, open source cross platform compiler from F# Software Foundation	
Rust	compiled. First, the Rust compiler does all the Rust specific stuff like type and borrow checking; in the end, it generates LLVM-IR. IR stands for intermediate representation and it's comparable to assembly, but a tiny bit more high level and most importantly: platform independent.	
Vb .Net	version 6 and above, both compiled and interpreted	interpreted
C#	compiled	
D	compiled	
Oz	ding52, Oz code can be compiled into command line executables. The compiled code is not native binary, but a shell script-wrapper with embedded Oz virtual machine bytecode.	ding52
Matlab		ding52, you can write code and just execute it from the IDE, without compilation.
R	an interface to compiled code, because all key routines are run in compiled code (through .C, .Call., .Internal, .Primitive interfaces, etc.) But does not compile	✔
Python		✔

Table 6: Assertion - 1

	Pre-Post conditions	Quantification	Pre-State Values	Global Assertions	Language Integration
Scala	ding54		✖	✖	import org.scalatest.Assertions._
Swift					
Falcon	✔	✔	✖	✔	import falcon
F#	✖	✖	✖	✖	open FsUnit [<AbstractClass>][<Sealed>]type Assert = class end
Rust	✔	✔	✔	✔	
Vb .Net	✖	✖	✖	✖	Debug.Assert Method System.Diagnostics Namespace Public NotInheritable Class Assert
C#	✖	✖	✖	✖	assert method in class Debug public static class Assert
D	✖	✖	✖	✖	✖
Oz	✔	✔	✔	✔	export Literals Assert
Matlab	✔	✔	✔	✖	✔, python, c,c++, C#, java, fortran
R	✔	✔	✔	✔	assertthat -assert_that() signal an error -see_if() returns a logical value, with the error message as an attribute. -validate_that() returns TRUE on success, otherwise returns the error as a string.
Python	✔	✔	✖	✖	assert method
	Pre-Post conditions	Quantification	Pre-State Values	Global Assertions	Language Integration

portion of code in a certain condition or to run a program with different version etc. For example, in case of error program should display a debug report so, in C, #ifdef will be used to define a debug. In HTML, different display sizes can be defined for different platforms like desktop, tablet, mobile etc. A compiler may be set to define different operating systems like windows, linux, mac etc. to compile the code accordingly or for javaascript versioning for different browsers.

3.8 File handling

File handling is used where data is required to be provided to the program from and to an external source, not using keyboard during program compilation. Data is stored in files that will be used by program during execution. Using file handling technique, data will be read from the file as soon as it requires without waiting for human user to input. Similarly information (output) will be saved to file and program execution will proceed without user to make any interaction like "Press any key to continue...". File handling has three steps 1) Opening a file: Opening a file for reading data from or writing data to an external file. 2) Reading/writing data: Reading data as input for program to store values in variables and operate on it or writing output to the file. 3) Closing a file: Closing the file once it's use is over. A file can be closed as soon as it's use is over or it can be closed at the end of program execution before closing the program.

3.9 Immutable

In muti-paradigm programming languages, an immutable object is an object whose state cannot be modified after it is created. This is in contrast to a mutable object (changeable object), which can be modified after it is created. In some cases, an object is considered immutable even if some internally used attributes change but the object's state appears to be unchanging from an external point of view. For example, an object that uses memorization to cache the results of expensive computations could still be considered an immutable object.

3.10 Mutable

A mutable object, in contrast to immutable, has data fields that can be altered. One or more of its methods will change the contents of the object, or has a property that, when written into, will change the value of the object. If you have a mutable object- the most similar one to String is StringBuffer inC#- then you have to make a copy of it if you want to be absolutely sure it would not change out from under you. This is why mutable objects are dangerous to use as keys in any form of Dictionary or Set- the objects themselves could change, and the data structure would have no way of knowing, leading to corrupt data that would, eventually, crashing the program. However, the contents can be changed with much more memory efficiency than making a complete copy. Generally, the right thing to do is use mutable objects while creating something, and immutable objects once creation is done. This applies to objects that have immutable forms, of course; most of the collections don't. It's often useful to provide read-only forms of collections, though, which is the equivalent of immutable, when sending the internal state of collection to other contexts- otherwise, something could take that return value. This could do something to to it, e.g corrupt it.

3.11 imperative control

Imperative control in programming languages allow to ex-

Table 7: Assertion - 2

	Security Levels	**enabling / disabling assertions**	**debugging support**	**inheritance**
Scala	✖	assume fail cancel succeed intercept exception assertDoesNotCompile assertCompiles assertTypeError withClue	Scala Debugger Intellij IDEA	subclass inherits any non-overridden methods in super class that contains asserts
Swift				
Falcon	✖	✖	falcon command line interpreter	✖
F#	✖	✖	.NET debugger System. Diagnostic. Debug. Assert	subclass inherits any non-overridden methods in super class that contains asserts
Rust	✖	✔	debug_assert! interpreter	✖
Vb .Net	✖	✖	VBasic .Net Debugger	✔
C#	✖	conditional compilation C# - .NETï£¡s Base Class Library (BCL) supports similar facility for C#, C++ and VBA.	C# debugger	subclass inherits any non-overridden methods in super class that contains asserts
D	✖	✖	✖	✖
Oz	✖	✖	✖	✖
Matlab	fatalAssertThat fataAssertWarning	✔ global NDEBUG; NDEBUG=true;	✔ matlab	✖
R	✖	✔ customized assert statements can also be written	✖	✖
Python	✖	✔	✔ python interpreter	✔
	Security Levels	enabling / disabling assertions	debugging support	inheritance

Table 8: Table of conditional compilation

	Conditional Compilation
Scala	scala.language.experimental.macros elidable
Swift	#if condition
Falcon	
F#	
Rust	#{[}cfg{]} for example #{[}cfg(foo){]}
Vb .net	#if?then?#else
C#	#if
D	debug { // ... conditionally compiled code ... } else { // ... code that is compiled otherwise ... }
Oz	
Matlab	#if (condition) {do something} #else {do something else} #end
R	
Python	

Table 9: Table of File Handling

Languages	File Handling
Scala	import scala.io.Source, Source.fromFile("any file")
Swift	Yes
Falcon	Yes
F#	open System.IO
Rust	io::Result<T>
Vb .net	Imports System.IO, FileStream = New FileStream("sample.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite)
C#	FileStream <object_name>= new FileStream(<file_name>, <FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);
D	import std.file; File file = File("test.txt", "w"); file.writeln("hello"); string s = file.readln();
Oz	Yes
Matlab	A = fscanf(fileID,formatSpec), [A,count] = fscanf(__) A = fscanf(fileID,formatSpec, sizeA)
R	list.files(file.path("F:", "git", "roxygen2")) file.create, file.exists, file.remove, file.rename, file.append, file.copy, file.symlink, file.link(from, to)
Python	file_object = open(?filename?, ?mode?)

Table 10: Mutable programming features in different languages

	Mutable
Scala	var maxValue = 100
Swift	var (firstNumber, secondNumber) = (10, 42)
Falcon	array=[1,2,3]
F#	let a = 1
Rust	let a = 1
Vb.net	Dim num1 as Integer = 1
C#	float PI = 3.14149
D	mutable int len = 1
Oz	{Browse {4+2} div 2}
Matlab	h = 6.626068e-34;
R	a<- 1
Python	var a = 1

plicitly define the execution order of program statements and expressions. Method or procedure is a sequence of control constructs. A method could be invoked, with or without passing parameters and its result could be returned to the caller. Body of method/procedure may contain imperative control expressions like ifï£¡else, switch statements and iteration statements like for, while, do etc.
In the true spirit of knowledge sharing, aiming to be practical, informative, and digestible, so much so that the reader could learn something, we'll enlighten advantages of Python. Under discussion of imperative control, this is most powerful language ever which provides a necessary facilities that a programmer may concern to while programming regarding control flow and functions calls. Next, we should be going to talk about two languages at once: C# and Visual Basic.NET. These are the two flagship languages for development on the Microsoft platform. We need to talk about them at the same time because although they do look different but these are by far the most popular. Both of them are object oriented. Both of them share the same characteristics. They are strongly typed and use garbage collection so we don't have to worry too much. These languages also include most of the features imperative control with less code and negligible effort is needed by programmer to maintain each and every thing. Table 8 describes comparison of imperative control structures among multi-paradigm programming languages.

3.12 Explicit concurrency

We say two activities are concurrent of they are executing in parallel or if they can be interleaved. The one which encapsulates a single concurrent activity, joining with a compulsory mutable state, is known as concurrency unit. Whether stream concurrency enables two or more concurrent activities to use each one end of stream. This could be thought as producer-consumer communication. In shared state, a shared data structure is modified by concurrent activities. Serialization or code locking ensures code segment to be executed only when permitted. In message passing concurrency, messages are exchanged among the concurrent activities. Message passing could be in synchronous (wait until receiving message) mode or asynchronous (not wait) mode or combination of the two modes. Message could be defined as data which is transferred among the activities. Order of message sending may not be important from the point of

underlying platform.
All the above multi-paradigm programming languages have disjunctions in many features that are important in context of programming. Very popular language for message passing concurrency is F# as it provides facility of MailBoxProcessor. Some that have good option to access feature of shared pool, are python and D. Although all the multi-paradigm languages provide facility of code locks, still python is best and also C# and VB.Net. Matlab and R do not facilitate concurrency. These languages neither provide any mechanism for message passing concurrency nor include feature of stream concurrency. Table 9 describes comparison of imperative control structures among multi-paradigm programming languages.

4. DISCUSSION AND ANALYSIS

Compiled/interpreted: Major advantage of compilation is the fast performance as it directly used the native code of the target machine and hence has the opportunity to apply quite powerful optimizations during the compile stage. Since the translation is done only once during the compilation, program only needs to be loaded and executed. Major advantage of interpreted is that ease of implementing logic especially for dynamic languages. Also there is no need to compile code and the programs can be executed directly. It is also easier to debug since programs can be executed side by side. Keeping this in mind, compiled languages shall be suitable for the intensive parts of an application requiring heavy resource usage whereas less intensive parts could be written in interpreted languages, e.g. interfaces, invoking the application, ad hoc requests or prototyping. If the programmer has to choose between speed and ease of programming, then the choice has to be made between languages opting for compiled or interpreted. A language having the facility of asserts provide the programmers with the ease of detecting errors that would have been impossible to catch using regular exception handling.
Asserts are a useful debugging tool. They help detect errors that might otherwise go undetected, detect errors sooner after they occur and also ensure that the statement about the effects of the code is true. The disadvantage of using asserts is reporting an error where none exists and failing to report a bug that does exist. Asserts are also not side-effect free. They also consume extra time and memory to execute. Assert is different from exception handling as occurrence of the exception may go unnoticed while asserts ensure one gets aware of the bug. Asserts are sometimes referred to as lazy exception handling.
Type safety helps programmer for the minimizing type errors occurs at run time. The programmer first assign types which is called binding. Binding can either be explicit or implicit. In explicit type binding, programmer specifically declare it type while implicit in type binding, compiler infer it's type by type inference method built in some languages tools. Some languages are strongly type, while some languages are weakly type. Strongly type languages guarantees that accepted programs are type safe. Weakly typed languages allow programs that contain type errors. Static type checking are efficient while dynamic checks slow down the programs.
Bound checking is found in strongly type languages guaranteeing type-safe execution by bound checking of array access. Array-bound check or index checking may give exceptions. Some researchers are eliminating bound checks as these checks slow the execution process. Some languages have options available to disable bound check while some programmers optimized their code by eliminating bound checking.

5. CONCLUSIONS

This paper discusses in detail various aspects of multi-programming paradigms and correlates different features with the programming languages lying in this paradigm. It has been concluded that multi-programming paradigm offers features from a broad spectrum making them more flexible, usable and applicable to diverse set of applications. This is also the reason for their increased popularity among the software community as programmers do not have to learn, shift or integrate works from different languages and get all the services from just one programming language belonging to the multi-programming paradigm. This frees the programmer from choosing the paradigm, but now only choice has to be made for the language only. Multi-programming paradigm itself offers a huge range of languages to choose depending on the features available. Some of the features common to all are type safety, assertion, file-handling and exception handling. While there is a variation in the way programming languages bound checking, meta-programming, compiled/interpreted, immutability, imperative control and explicit concurrency. This work shall act as baseline for any future work done in enhancing the understanding of the how various features correlate in different programming languages lying in the multi-programming paradigm.

6. FUTURE WORK

We plan to extend our work by studying, analyzing and comparing more languages as well as as with more features. We plan to come up with a more exhaustive study of the features of the languages in multi-programming languages. We also plan to list and justify the popular versus usable features for the multi-paradigm programming languages for the contemporary times.

Table 11: Imperative control of multi-paradigm programming languages

	Method/Procedure Parameters	Method/Proc Return	Control Statement	Iteration
Scala	By name or function pointer	Positional, by val	If (x>y) {max= x;}	While, do-while, for
Swift	By name or function pointer	Positional, by val	If x>y {max= x;}	For-in, while
Falcon	By name or function pointer	Positional, by val	If x>y {max= x;}	For, while
F#	By name	By val, by ref	If (x>y) then num1	Whileï£¡do, forï£¡in/to/downto
Rust	By name or function pointer	Positional, by val	If x>y {max=x;}	Loop, while, for-in
Vb .Net	By name or function pointer	By value, ref, name	If x>y then x endif	Whileï£¡end, do-loop,
C#	By name or function pointer	By val, ref, name	If (x>y) { max=x; }	While, do-while, for, foreach
D	By name or function pointer	Positional, by val	If (x>y) {max=x;}	While, do-while, for
Oz	By name or function pointer	Positional, by val	if x>y then x end	For ï£¡ doï£¡end
Matlab	By name or function pointer	Positional, by val	If (x>y) x end	While, for
R	By name or function pointer	Positional, by val	If (x>y)	Repeat, While, for
Python	By name or function pointer	Positional, by val	If x>y: max=x	While, for

Table 12: Explicit concurrency of multi-paradigm programming languages

	Conc. Unit	Stream Concurrency	Message passing	Code locking
Scala	SynchVar	AKKA	Actor	Java library
Swift	nsthread	nsstream	✖	Lock.swift
Falcon	✔	✔	✔	-sync y
F#	LockObject	Threading	✖	LockedCounter
Rust	Arc, mutex	Thread	rc	
Vb .Net	thread	threading	Thread pool	Synclock
C#	Parallel.for	Threading	monitor	Lock()
D	tid	Std.concurrency	Shared()	core.sync.mutex
Oz	thread	✖	Lck p	✖
Matlab	✖	✖	✖	Lock, mutex
R	✖	✖	✖	Lock
Python	✖	Threading	Thread pool	Lock, semaphore

7. ADDITIONAL AUTHORS

8. REFERENCES

[1]

[2]

[3] MS Windows NT kernel description. Accessed: 2010-09-30.

[4] Multi-paradigm programming language. 2013.

[5] S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, pages 875–903, 2005.

[6] Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 285–298, New York, NY, USA, 1993. ACM.

[7] M. Jazayeri C. Ghezzi. Programming language concepts. 1997.

[8] Robert G. Clark. Comparative programming languages. 2000.

[9] Lori A Clarke and David S Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[10] Frans Coenen. Characteristics of declarative programming languages. 2014.

[11] Cleenewerck T. Ganther, S. design principles for internal domain-specific languages: a pattern catalog illustrated by ruby. *n: Proceedings of the 17th International Conference on Pattern Languages of Programs (PLoP'10)*, 2010.

[12] Wider A. Scheidgen M. George, L. Type-safe model transformation languages as internal dsls in scala. *In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT'12)*, 13:160–175, 2012.

[13] Michael Hanus. Multi-paradigm declarative languages. 2007.

[14] Charles Antony Richard Hoare. Assertions: A personal perspective. *IEEE Annals of the History of Computing*, 25(2):14–25, 2003.

[15] T. El-Ghazawi J. Urbanic K. Ebcioglu, V. Sarkar. An experiment in measuring the productivity of three parallel programming languages. *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing ser. P-PHEC*, 6:30–37, 2006.

[16] Poul-Henning Kamp. My compiler does not understand me. *Communications of the ACM*, 55(7):51–53, 2012.

[17] F. Shull-S. Asgari V. Basili J. K. Hollingsworth M. V. Zelkowitz L. Hochstein, J. Carver. Parallel programmer productivity: A ca,se study of novice parallel programmers.

[18] Kurt Normark. Overview of the four main programming paradigms. *ACM SIGSOFT Software Engineering Notes*, 2012.

[19] Kurt Normark. Overview of the four main programming paradigms. 2012.

[20] I. Neamtiu P. Bhattacharya. Assessing programming language impact on development and maintenance: A study on c and c++. *Proceedings of the 33rd International Conference on Software Engineering ser. ICSE*, 11:171–180, 2011.

[21] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33:23–29, 2000.

[22] Andrew Lumsdaine-Jeremy Siek Jeremiah Willcock Ronald Garcia, Jaakko Jarvi. A comparative study of language support for generic programming. 2003.

[23] K. Soares da Silveira S. Nanz, S. West.

[24] Moha N. Baudry B. Jï£¡zï£¡quel J.M. Sen, S. Meta-model pruning. *In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (Models)*, pages 32–46, 2009.

[25] E. Syriani. A multi-paradigm foundation for model transformation language engineering. 2011.

[26] D. Lo L. Jiang L. Roeillre T. E. Bissyand, F. Thung. Popularity interoperability and impact of programming languages in 100000 open source projects. *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference ser. COMPSAC*, 13:302–312, 2013.

[27] G. Garretn V. Pankratius, F. Schmidt. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java. *Proceedings of the 2012 International Conference on Software Engineering ser. ICSE*, 12:123–133, 2012.