The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.82 e-ISSN: 2251-7545

Multi-paradigm Programming with Modern Languages

Shrinidhi R. Hudli
Department of Computer Science
M.S. Ramaiah Institute of Technology
Bangalore, India

Raghu V. Hudli
ObjectOrb Technologies Pvt. Ltd.
Bangalore, India

***Abstract*—Most modern programming languages support multiple programming paradigms. For example, C++ supports procedural and object-oriented programming. Java supports mostly object-oriented programming, though one could stretch its features to write procedural programs. Languages like Ruby Python, Groovy, and Scala, among others, support functional programming, procedural programming, and object-oriented programming. Our interest is in examining the features pertaining to functional programming and object-oriented programming. Specifically, out interest is in the correspondence between closures in functional paradigm and objects. In this paper we show that closures and subsumed by objects. We demonstrate subsumption using structural analysis.**

Keywords- object-oriented programming, functional programming, closures, objects

I. INTRODUCTION

Earlier programming languages were designed to support specialized programming paradigms. For example, LISP was designed to implement functional programming, while C supported procedural style and Prolog supported logic programming. Many languages designed since 1980s support multi-paradigm programming. For example C++ supports procedural and object-oriented programming. While Java has a much more aligned with object-oriented programming compared with C++, it is possible to write procedural style programs with Java. Modern languages like JavaScript, Ruby, Python, Scala, Groovy, etc. support multiple paradigms. One can write procedural, functional and object-oriented programs in modern languages.

It has been established that object-oriented designs and programs are needed to tackle the complexity of modern software. Object-oriented design lends itself to clean separation so of concerns, allocation of responsibilities and structural decomposition of software to ensure maintainability and extensibility [1,2,3]. The main idea with object-oriented design is to identify key abstractions and their relationships, including subtypes besides other relationships such as associations, etc. There is a vast body of knowledge containing patterns and principles [1,4] that designers can utilize to arrive at object-oriented designs that are efficient and robust.

Though functional programming paradigm is old, there is a renewed interest in functional programming. The functional paradigm strives for decomposition of software into set of functions. Functions in the functional paradigm are functions in the mathematical sense – they map values from a domain to a range. The domain can be formed using Cartesian products of other domains. The distinguishing feature of functions is they return a single value for the given inputs and do not cause side effects on the parameters. The immutability of parameters has sparked renewed interest in modern programming, especially for high-performance parallel programming. Languages like JavaScript, Ruby and Groovy also use functional constructs for cleaner and elegant programming constructs.

In this paper we specifically examine closures, an important construct of functional programming and its relationship to objects in a multi-paradigm language. We show that objects can be uses to achieve the same functionality of closures. Closures have been popular in languages like JavaScript, Ruby and Groovy. There was considerable push to incorporate closures in Java 7, but was not included. Now, there is renewed effort for inclusion in Java 8. We ask the question if closures from the functional paradigm provide only syntactic sugar in multi-paradigm programming languages.

II. OBJECT-ORIENTED PROGRAMMING

After nearly two decades of pedagogy and construction of software based on object-oriented programming (OOP) principles, OOP is quite often the programming paradigm of choice by default.

The key underpinning of OOP is abstraction of behavior. Ensuring that an object represents a single abstraction/responsibility [1] moves us towards building maintainable software. Behavior is modeled using methods that be invoked on objects. Satisfying Single Responsibility Principle [1] ensures that methods will be cohesive – that is all methods of the class representing the object perform related actions.

Encapsulation in OOP ensures that the data needed for methods is available for them. This is typically implemented using instance data of objects. We give two examples of classes in Ruby.

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.82 e-ISSN: 2251-7545

```
class Player
    def initialize(name)
        @name = name
    end
    def set_club(club)
        @club = club
    end
    def get_name
        @name
    end
    def get_club
        @club
    end
end
```

```
class Club
    def initialize(name)
        @name = name
        @players =  []
    end
    def get_name
        @name
    end
    def add_player(player)
        @players << player
    end
    def number_of_players
        @players.length
    end
end
```

Classes provide the abstraction in OOP. In the above example, the Player class provides the abstraction that a player has a name and a club. The Club provides the abstraction that clubs also have names and a collection of players. In the abstraction we have modeled here, a Player can belong to only one Club; this is a minor detail that does not affect the topic of discussion of this paper. In another object-oriented language the structure of the classes would be equivalent. We do not have an explicit method to set the name of the player or club as the name is associated with the object at construction time in the initialize method (aka constructor in Java and C++).

The next key feature of OOP that we need for the purpose of discussion in this paper is that of encapsulation. The data needed for the behavior is part of the object. In the example here the attribute name, denoted @name in Player and Club is part of each of the objects and is distinct for each instance. Similarly @club is the Club attribute on the player. The collection of players needed by Club is part of the Club object. Hence it is possible to implement methods number_ of_players without passing a collection of Players to a Club object.

Subtyping or inheritance is another feature of OOP that we need for this paper. Subtypes provide a powerful mechanism to not only reuse classes, but to construct programs with abstractions and substitute abstractions by subtypes. It suffices here to state that subtypes cannot enforce stronger type requirements that the types that they extend. They have to support Liskov Substitutability Principle [1,5].

We now define a subtype of a Player called Batsman as shown below. In Ruby, type extensions are indicated by "<" character.

```
class Batsman < Player
        attr_accessor :batting_position
        def initialize(name)
            super(name)
        end
end
```

We have introduced a new attribute called batting _position that is needed for the Batsman class. Since the Player class was more abstract the specific attribute indicating where in the batting lineup a player was is irrelevant, but is needed for the Batsman abstraction. The attr_accessor is a Ruby detail to indicate that batting_position is attribute that can be set and queried.

The key thing to note here is that Batsman not just reuses the Player abstraction, but is substitutable wherever Player abstraction is used. For example, a Batsman object can be added to a Club using the add_player method defined in the Club class.

III. FUNCTIONAL PROGRAMMING

Classical functional programming (FP) is based on and derived from Lambda Calculus [6]. It is now a very well established fact that Lambda Calculus is equivalent to any computable function. Lambda Calculus, originally proposed by Alonzo Church [7] is incredibly succinct for its power. It has only three constructs

<expression> ::= <name> | <function> | <application>

A name is a sequence of non-blank characters.

A function is an abstraction and definition of a function. It has two parts, a name and a body, the name is usually a variable and body is a lambda expression.

Examples of functions are

λx.x+1 which is the successor function to increment a value

λx.x is an identity function

Once functions are defined, they can be invoked. Invocations are called application in Lambda Calculus. The syntax for application is

<application> ::=

(<function expression> <argument expression>)

An example of application is

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.82 e-ISSN: 2251-7545

(λx.x+1 2) which will yield 3, since 2 is applied to the increment Lambda expression

while

(λx.x λa.λb.b) will yield λa.λ b.b, since the λa.λb.b Lambda expression is applied to the identity Lambda expression.

Languages like Ruby provide a clean syntax for defining Lambda expressions. For example a successor function can be created using the following syntax in Ruby

```
succ = -> x {x + 1}
```

or

```
succ = lambda {|x| x + 1}
```

This creates a Lambda expression called succ. The application is simply a call on the Lambda expression, using a method called *call* as below:

```
succ.call(2)
```

JavaScript also supports creation of Lambda expressions. Here is an example.

```
function createLambda(){
    function succ(x) {
      alert(x+1);
    }       return succ;
}
```

The application is simply a function call to the JavaScript function that is returned. The example below illustrates it.

```
var mySucc =  createLambda()
mySucc(2)
```

It should be clear to the discerning reader that such constructs can be created even in languages like C using function pointers. However, they are very cumbersome.

In Lambda expressions, variables can be bound or free. A variable is bound if it is bound in an expression. For example in λx.x+1, x is a bound variable as the variable x is bound to the Lambda expressionλ.y x+1. However, y is a free variable in orλxin.x+y.

The application of functions that have free variables present interesting situations. Lambda expressions can be created with the free variables *frozen* to an execution context. Such Lambda expressions are called **closures**. The Ruby procedure below creates a Lambda Expression with a free variable y. The variable is λxfree.x+y inthatthe Lambda Expression createIncrementingProc creates, but is bound to the environment or context in which it is created when createIncementingProc is called. The Ruby Proc object creates a Lambda expression.

```
def createIncrementingProc(y)
    Proc.new {|x| x + y}
end
```

Now using createIncrementingProc, we can create distinct execution environments. inc_by_10 creates a closure λx.x+y with the free variable y frozen to 10 and inc_ by_20 createsλx.x+y,aclosurewith free variable f rozen to 20.

```
inc_by_10 = createIncrementingProc(10)
inc_by_20 = createIncrementingProc(20)
```

Closure application now requires only value for the bound variable xλx.x+yinthe.AsLambdabefore, expression application requires calling the method called *call*

```
inc_by_10.call(3)
inc_by_20.call(3)
```

The first call to Lambda expression evaluates to 13, while the second to 23. Closures in JavaScript have identical behavior, save for the syntax. The same is true for Python and other languages.

Closures are quite useful. In languages like Ruby and Groovy, they are often used to pass as arguments to control structures or iterators. They have also been used as constructs for event handling and implementing callbacks. It is quite convenient to register a callback closure with a defined execution context and the caller need not be aware of the free variables in the execution context.

IV. CLOSURES AND OBJECTS

Closures have been used in some languages to also implement objects. Our view from application programming is opposite. For a programmer, objects are far more sophisticated than closures and what can be achieved by closures, can be achieved by objects too. We will also see in this section, that objects can go beyond closures and also support function subtyping.

If closures are dissected, then we see two parts to closures. The first part is the execution context where the free variables are frozen; for example setting the value of y to 10 or 20 based on how createIncrementingProc was called. The second part is the Lambda expression itself. While Lambda expressions can be replaced by methods. For example,

```
succ = lambda {|x| x + 1}
```

is equivalent to

```
def succ(x)
        x + 1
```

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.82 e-ISSN: 2251-7545

```
end
```

So replacing Lambda expressions by equivalent functions or methods is straightforward. We now have to consider creation of the execution context for the Lambda expressions to implement closures. Object attributes provide a convenient and flexible mechanism to provide execution context. We will illustrate the idea through an example.

```
class ClosuresWithObjects def
    initialize(y)
        @y = y
    end
    def call(x) x +
        @y
    end
end
```

The ClosuresWithObjects class defines a single method called invoke which takes the bound variables of the createIncrementingProc closure. The free variable y is frozen in the initialize function. When an object of ClosureWithObjects class is created, the execution environment is frozen exactly as in closures. We now examine the creation of closures using objects and invocation of closures. Closures are created by just instantiating objects, unlike calling of procedures in the previous example.

```
inc_by_10 = ClosuresWithObjects.new(10) inc_by_20 =
ClosuresWithObjects.new(20)
```

Now inc_by_10 and inc_by_20 are object instances. These instances already have execution context created just like in closures. Application of closures is just invocation of the call method on these object instances, as below.

```
inc_by_10.call(3)
inc_by_20.call(3)
```

The behavior of the objects is identical to closures. Structurally they are similar, though there are syntactic differences between closures and objects. Having demonstrated that objects provide behavioral equivalence of closures, we will now examine the advantages of using objects over closures. There are two significant benefits of using objects over closures even when closures are needed.

In the case of closures the execution context is frozen, but with objects it is possible to reset or change the execution context, which can be convenient in some cases. We simply provide a method or methods to modify the execution context. For example by adding a set_x method to the ClosuresWithObjects, we can set new values of x. This allows reuse and/or reconfiguration of closures implemented with objects.

A second and more important benefit of implementing closures with objects is the support for function subtyping [5]. The type of a function can be defined on the basis of the domain and range it operates on. If T1 is the type of the function that maps from D1 to R1, then we have

T_1: D_1 -> R_1

A function of type T_2 is a subtype of T_1 if

T_2: D_2 -> R_2 such that D_1 <: D_2 and R_2 <: R_1

where <: indicates subtype relationship.

The range types can be covariant with subtyping while the domain types are contravariant. These are consistent with Liskov Substitutability Principles for objects. Function subtyping can be easily implemented by subclassing.

The increment closure maps from float -> int. If we wanted a function subtype that incremented from double -> int, then it is easily obtained by subclassing and overriding the call method. Of course, languages like Ruby support ducktyping, covariance of ranges and contravariance of domains are easily supported. But statically typed languages like Java and C++ can provide this behavior quite easily.

Objects are far more general and powerful than closures. Objects can provide the full semantics of closure with the extension of providing subtyping of closures and providing an execution context that is mutable and not frozen.

However there is interest in modern object-oriented programming languages to provide support for Lambdas and closures. Some example are Pythton, Ruby, Java 7 that provides Lambda and Java 8 is debating inclusion of closures, and C++0x has support for Lambda. In our opinion these language extensions only provide syntactic sugar without really extending the expressive power of the core language. As demonstrated, what Lambdas and closures provide, objects are capable of providing the same semantics and more.

The International Journal of Soft Computing and Software Engineering [JSCSE], Vol. 3, No. 3, Special Issue:
The Proceeding of International Conference on Soft Computing and Software Engineering 2013 [SCSE'13],
San Francisco State University, CA, U.S.A., March 2013
Doi: 10.7321/jscse.v3.n3.82 e-ISSN: 2251-7545

V. CONCLUSIONS

We examined multi-paradigm programming in modern languages with the special focus on closures from functional paradigm and objects. There is renewed interest in bringing closures to Java 8 standard. Our analysis shows that objects provide all the semantics of closures and more. Closures, in our opinion, provide syntactic sugar, while objects provide additional features of resetting or changing execution context and function subtyping.

REFERENCES

[1] Robert C. Martin, Agile Software Development: Principles, Patterns and Practices, Prentice Hall, 2002
[2] Eric Evans, Domain Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003
[3] Grady Booch, Object-Oriented Analysis and Design with Applications, Addison-Wesley, 2007
[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns, Addison-Wesley, 1994
[5] Benjamin C. Pierce, Types and Programming Languages, MIT Press, 2002
[6] Greg Michaelson, An Introduction of Functional Programming Through Lambda Calculus, Dover, 2011
[7] Alonzo Church, The Calculi of Lambda-Conversion, Princeton University Press, 1941