San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

1-1-2008

JavaFX as a Domain-Specific Language in Scala / Groovy

Sadiya Hameed
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Recommended Citation
Hameed, Sadiya, "JavaFX as a Domain-Specific Language in Scala / Groovy" (2008). *Master's Projects.* Paper 17.

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

JavaFX as a Domain-Specific Language
in
Scala / Groovy

A Project Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

By

Sadiya Hameed

May 2008

© 2008

Sadiya Hameed

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Cay Horstmann

Dr. Jon Pearce

Dr. David Taylor

APPROVED FOR THE UNIVERSITY

ABSTRACT

JAVAFX AS A DOMAIN-SPECIFIC LANGUAGE
IN SCALA / GROOVY

By Sadiya Hameed

Domain-Specific Languages (DSLs) are optimized for developing applications in a particular domain. JavaFX is such a DSL for creating rich graphical user interfaces.One method to create a DSL is to implement it in an existing language. This offers the advantage that existing users of the language do not need to learn a new language to work in the DSL. Scala and Groovy are two upcoming Java based languages which boast a feature set that can be used to extend existing languages and facilitate DSL creation. In this project my goal was to evaluate the ability of Scala and Groovy to be DSL hosts. To this end, I implemented my own JavaFX like DSLs in Scala and Groovy and assessed their capability for constructing a DSL.

ACKNOWLEDGEMENTS

First of all I would like to thank Dr. Cay Horstmann for trusting me with his idea. Without his guidance, suggestions and support this project would not have been possible. More than an advisor he has truly been a mentor.

I would also like to extend my heartfelt appreciation to both Dr. Jon Pearce and Dr. David Taylor for their input, suggestions and time. And also for agreeing to be on the project committee of a completely unknown student.

On a personal note, I would like to thank my husband Umair for bearing with me throughout the process and for all the encouragement, support and help. Especially for all the hours you lost not playing videogames just so that I would not get tempted.

TABLE OF CONTENTS

Introduction 1
1. Domain-Specific Languages 2
2. JavaFX 2
  2.1. List of Important Features 3
    2.1.1. Incremental Dependency-Based Evaluation 3
    2.1.2. dur Operator 5
    2.1.3. do and do later 6
  2.2. JavaFX Progress (Interpreted vs. Compiled) 7
3. Host languages 8
  3.1. Scala 8
    3.1.1. Operators as Valid Identifiers 8
    3.1.2. Single Parameter Methods as Infix Operator 8
    3.1.3. Methods Without Arguments 8
    3.1.4. Properties 9
    3.1.5. Functions and Closure 9
    3.1.6. Case Classes and Pattern Matching 10
    3.1.7. Views 11
  3.2. Groovy 12
    3.2.1. Parentheses-less Methods and Named Parameters 12
    3.2.2. Closure 12
    3.2.3. Categories 13
    3.2.4. DelegatingMetaClass 13
    3.2.5. ExpandoMetaClass 14
4. Rich Graphical User Interface DSL 15
  4.1. DSL in Scala 15
    4.1.1. Incremental Dependency-Based Evaluation (Bind) 15
      4.1.1.1. Bound Property 15
      4.1.1.2. Bound Swing Components 17
      4.1.1.3. Bound Bean Property (An Alternate Approach to bind) 20
      4.1.1.4. Bind with Expressions 22
    4.1.2. Animate Feature 25
    4.1.3. `doOutside` and `doLater` 27
  4.2. DSL in Groovy 29
    4.2.1. Incremental Dependency-Based Evaluation (Bind) 29
      4.2.1.1. Bound Property 29
      4.2.1.2. Bound Swing Components 30
      4.2.1.3. Bind with Expressions 33
    4.2.2. Animate Feature 34
    4.2.3. `doOutside` and `doLater` 35
5. Evaluation 38
  5.1. DSL Comparison 38
    5.1.1. Bind 38
      5.1.1.1. Bound Property 38

5.1.1.2. Bound Swing Components 39
5.1.1.3. Bind with Expressions 39
5.1.2. Animate Feature 41
5.1.3. Concurrency Features 41
5.2. Comparison of Language Features 42
6. Conclusion 44
References 46

LIST OF FIGURES

Figure 1. Screenshots of `dur` operator demo at 0 seconds and 6 seconds 5
Figure 2. Screenshot of `Swing` component binding demo 19
Figure 3. Screenshot of Bind with Expressions demo application 22
Figure 5 Screenshot of Animation demo at 1 second and 3 second 25
Figure 6 Screenshots from model-view binding demo. 31

Introduction

Domain-specific programming languages (DSLs) are designed for a particular problem domain and promise substantial expressiveness and ease of use in their specialized area over general-purpose programming languages. JavaFX is such a domain-specific language aimed at speedy development of rich user interfaces for Java. The core of JavaFX is JavaFX Script, a declarative scripting language with a high degree of interactivity with Java classes.

It seems unfortunate that JavaFX is yet another language. The modern trend is to provide DSLs inside a larger host language to facilitate programmers to employ their knowledge of existing languages when using a DSL as opposed to learning a new language. The goal of the project is to examine the feasibility of mimicking the functionalities provided by JavaFX as a DSL in Groovy and Scala languages and to reason about the suitability of these two languages as DSL hosts.

Scala claims to have been invented for just this purpose. "Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries:

- any method may be used as an infix or postfix operator, and
- closures are constructed automatically depending on the expected type (target typing).

A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities." [5]

Groovy has had practical success in providing DSLs for XML builders, ORM, etc. and its builders claim that it is particularly well suited for writing a DSL. "Groovy provides various features to let you easily embed DSLs in your Groovy code. e.g.

- you can create your own control structures by passing closures as the last argument of a method call
- it is possible to add dynamic methods or properties (methods or properties which don't really exist but that can be intercepted and acted upon) by implementing GroovyObject or creating a custom MetaClass, etc." [10]

The project aim was to create a rich graphical user interface (GUI) DSL in the two languages to facilitate evaluation and comparison of their ability to be DSL hosts. To this end, I choose the most valuable features of JavaFX that are its raisons d'être. This report gives a comprehensive view of the project work. It starts with a description of the features of JavaFX which I have implemented in my DSL. Next, it lists features of Scala and Groovy which facilitate addition of dynamic behavior in these languages. The report then gives a detailed description of the DSLs created in Scala and Groovy. It also details the implementation and usage of the DSL features. In the end I have evaluated both Scala and Groovy on their capability to be DSL hosts based on my experience with them during the creation of the rich GUI DSLs.

1. Domain-Specific Languages

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They offer substantial gains in clarity and usability over general-purpose programming languages, in their domain of application. They can be formally defined as:

"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [9]

DSL development is hard, requiring both domain knowledge and language development expertise. According to Mernik et al., one way to design a DSL is to base it on an existing language. Possible benefits are easier implementation and familiarity for users, but the latter only applies if users are also programmers in the existing language. One of the approaches to designing a DSL listed according to Mernik is to take an existing language and extend it with new features that address domain concepts. In most applications of this pattern, the existing language features remain available. The challenge is to integrate the domain specific features with the rest of the language in a seamless fashion. This is the approach I have employed in creating the JavaFX like DSL in existing languages of Scala and Groovy.

2. JavaFX

JavaFX is a domain-specific language for content developers to create rich media and interactive content. The core of JavaFX is JavaFX Script which is a declarative and statically typed scripting language. It supports first-class functions, list-comprehensions, and incremental dependency-based evaluation. It can make direct calls to Java APIs. Since JavaFX Script is statically typed, it has the same code structuring, reuse, and encapsulation features (such as packages, classes, inheritance, and separate compilation and deployment units) that make it possible to create and maintain very large programs using Java technology.

JavaFX was introduced at JavaOne 2007 and until very recently was an interpreter based language. Currently efforts are underway to port it to a compiler and therefore the language syntax and features are influx. The features and syntax described in this report pertain to the interpreted version of JavaFX. Section 2.2 gives an overview of some of the changes that have occurred so far in the compiled JavaFX version which is currently under development.

Although the official name of the scripting language is JavaFX Script it is generally referred to as JavaFX since it is at the core of the JavaFX family. This report also uses the name JavaFX to refer to the scripting language.

2.1. List of Important Features

Following is a list of important JavaFX features. The selection is based on the key features of JavaFX that facilitate Graphical User Interface (GUI) development. These are the same features that I have provided in my DSL. Since JavaFX is in its nascent stages, it has very limited documentation. Therefore, most of the feature descriptions have been taken from the JavaFX project website [8].

2.1.1. Incremental Dependency-Based Evaluation

JavaFX allows attribute values to be dependent on other attributes, or expressions involving other attributes. An update in any attribute automatically updates the values of all attributes that are directly or indirectly dependent on it. This feature can be thought of in the same way as the automatic updating of a spreadsheet cell, that is attached to a formula, when other cells from the formula change their value. It is especially useful for GUI development where maintaining model and view attributes in sync usually requires complex procedural logic.

Consider the following model-view based example where the value of the view's **`title`** is dependent upon the model's **`greeting`** attribute.

```
import javafx.ui.*;

class HelloWorldModel {
   attribute greeting: String;
}
var model = HelloWorldModel {
   greeting: "Hello World"
};
Frame {
   title: bind "{model.greeting} JavaFX"
   width: 200
   content: TextField {
      value: bind model.greeting
   }
   visible: true
};
```

Listing 2-1 Sample JavaFX code for bind between model and view

Initially the Frame's **`title`** and TextField's **`value`** are set by the model's **`greeting`** attribute. When the user updates the TextField's **`value`** the model's **`greeting`** attribute is automatically updated (which in turn updates the **`title`**). The next example show the binding of a variable to an expression involving other variables.

```
var ItemCost = 50;
var NumberOfItems = 10;
var Total = bind ItemCost * NumberOfItems;
assert Total == 500;
```

Listing 2-2 Sample code for JavaFX bind with expression

In the code above, whenever the value of **`ItemCost`** or **`NumberOfItems`** changes, the **`Total`** is updated. Binding with expressions is unidirectional so a change in **`Total`** will not reflect on the variables it is bound to.

Interpreter based JavaFX has two kinds of methods, functions and operations. It allows binding to work with both of them. Functions fall in the purely functional aspect of JavaFX. The body of a function may only contain variable declarations and a return statement. Conditional operations or loops are not allowed inside the body of a function. Functions are designed to be used with binding. Because functions incrementally update their results whenever either their arguments or referenced variables change, binding to a function works just as well as binding to a single attribute. Operations, on the other hand are more like methods of Java. They can contain any statement such as conditionals or loops etc. Unlike a function, changes to local variables inside an operation do not trigger incremental evaluation. Incremental evaluation is not performed inside the body of an operation except for expressions explicitly prefixed by **`bind`** [8].

In the compiled version of JavaFX, its creators are doing away with methods and introducing concept of bound vs non-bound functions. The bound functions are preceded by the bound keyword [17]. The two concepts and their details are still being worked out. Binding with functions is not a part of my DSL.

2.1.2. dur Operator

The **dur** operator (**dur** stands for duration) allows you set a variable to an entire range of values over the span of time. When assigning any value with an array of possible values with the **dur** operator, JavaFX will create a background thread which will assign the range of values into the variable. By binding the location, size, or transformation of a graphical element, this can create a smooth animation.

The documentation for the **dur** operator is sparse and the JavaFX Language Reference (for the interpreter or the compiler) does not have any mention of the **dur** operator as yet. The demo below has been taken from the weblog of JavaFX's creator Chris Oliver. [11]

```
import javafx.ui.*;
import javafx.ui.canvas.*;
Rect {
     fill: purple
     height: 50
     width:  50
     var x = 0
     var y = 0
     x: bind x
     y: bind y
     onMouseClicked: operation(e) {
          for (i in [0..100] dur 3000){
               x = i;
          }
          for (i in [0..100] dur 3000) {
               y = i;
          }
     }
}
```

Listing 2-3 Animated rectangle demo for JavaFX `dur` operator

In the above application, a rectangle moves a 100 pixels to the right in three seconds and then a 100 pixels down in another 3 seconds. The animation starts on a mouse click.

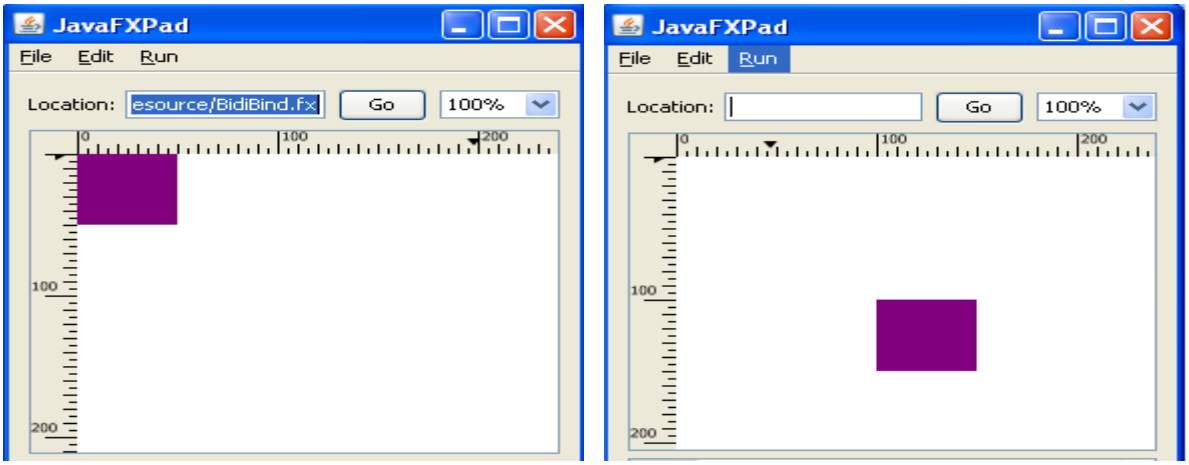

Figure 1. Screenshots of `dur` operator demo at 0 seconds and 6 seconds

2.1.3. do and do later

To understand the **do** and **do later** features of JavaFX, a little background on threads and Swing is required. Since Swing is not thread safe so all updates to the UI components should be done within the Event dispatch thread (EDT). EDT is a background thread that handles all the GUI related events like redrawing components or I/O events on the user interface. On the other hand if a task takes a long time, it should be done in a separate thread. If an extensive task is handled in the EDT the UI will become non-responsive until the task is finished. If your application needs to update the UI it can hand off that task to the EDT by calling the **invokeLater** method in the **EventQueue** class. This method puts the task in the EDT queue and returns. The task is then executed asynchronously. [12]

Normally all JavaFX code executes in the EDT, but with the **do** feature JavaFX allows an easy way for programmers to execute a portion of their code in a separate thread by placing it inside the body of a **do** statement. Sun's tutorial [16] mentions the following example where the reading from a URL is done inside the **do** statement and the GUI events continue to be processed during the read.

```
import java.net.URL;
import java.lang.*;
import java.io.*;

//executes in the AWT event dispatch thread (EDT)
var result = new StringBuffer();

do {
     //executes simultaneously in the background thread
     var url = new URL("http://www.foo.com/abc.txt");
     var is = url.openStream();
     var reader = new BufferedReader(new InputStreamReader(is));
     var line;
     while (true) {
          line = reader.readLine();
          if (line == null) {
               break;
          }
          result.append(line);
          result.append("\n");
     }
}

// now back in the EDT
System.out.println("result = {result}");
```

Listing 2-4 Sample code for usage of JavaFX do statement

The **do** statement has another form called the **doLater** which allows for asynchronous execution of tasks in the EDT. It is similar to the functionality provided by **java.awt.EventQueue.invokeLater()** discussed above.

2.2. JavaFX Progress (Interpreted vs. Compiled)

JavaFX is being converted from being interpreted to being compiled and in the process some changes to features and syntax are taking place. Below are a few of the changes that affect the features discussed in this report.

1. The compiled syntax merges the concept of function and operation into function. Functions are no longer restricted and have the same capabilities as the old operation concept. The working of bind on the new function concept is still being worked out.

Interpreted:
```
class Foo{
  function times2(x) { return x * 2; }
  operation print(s) { System.out.println(s); }
}
```
Compiled:
```
class Foo {
  function times2(x) { return x * 2; }
  function print(s)  { System.out.println(s); }
}
```

2. Bi-directional binding needs to be explicitly declared using the “with inverse” syntax. Otherwise unidirectional binding is assumed.

Interpreted:
```
TextField {
  value: bind model.firstName
}
```

Compiled:
```
TextField {
  value: bind model.firstName with inverse
}
```

3. In the interpreted version the initial value of an attribute was set outside of the class body. Attribute initial values are now set as part of the declaration, as in Java.

Interpreted:
```
class Foo {
  attribute bar: Boolean;
}
attribute Foo.bar = true;
```

Compiled:
```
class Foo {
  attribute bar: Boolean = true;
}
```

3. Host languages

3.1. Scala

Scala fuses object-oriented and functional programming in a statically typed language that runs over the Java Virtual Machine. Scala programs resemble Java programs in many ways and they can seamlessly interact with code written in Java. Scala is designed to be extensible and provides mechanisms to add new language constructs. Below is a list of Scala features that I have used for the creation of my DSL.

3.1.1. Operators as Valid Identifiers

Scala treats operator names as ordinary identifiers. An identifier is either a sequence of letters and digits starting with a letter, or a sequence of operator characters. The precedence of a user-defined infix operator is based on the first character of the operator name. For those operators whose first character is an operator in Java, the precedence is the same as in Java [14].

3.1.2. Single Parameter Methods as Infix Operator

Any method which takes a single parameter can be used as an infix operator in Scala. This is good for syntactic sugar which is important for DSL creation. Below is an example class given in the Scala tour. [5]

```
class MyBool(x: Boolean){
  def AND(that: MyBool): MyBool = if (x) that else this
  def OR(that: MyBool): MyBool = if (x) this else that
  def negate: MyBool = new MyBool(!x)
}
```

Now, we can use **AND**, **OR** and **negate** as infix or postfix operators.

```
def NOT(x: MyBool) = x negate;
def XOR(x: MyBool, y: MyBool) = (x OR y) AND NOT(x AND y);
```

3.1.3. Methods Without Arguments

Scala allows methods without arguments to be declared and called without the parentheses as opposed to methods with zero arguments that require parentheses in both cases. The Scala Tutorial [15] shows how accessors in a class can be declared and used in this way.

```
class Complex(real: double, imaginary: double){
  def re = real        //re and im are methods without arguments
  def im = imaginary
  override def toString() = "" + re +(if(im < 0) "" else "+")+ im + "i"
}
```

3.1.4. Properties

For every definition of a variable **`var x: T`** in a class, Scala defines setter and getter methods. Every mention of the name **`x`** in an expression is then interpreted as a call to the parameterless method **`x`**. Every assignment **`x = e`** is interpreted as a method invocation **`x_=(e)`**. This treatment of variable accesses as method calls makes it possible to define properties (in the C# sense [18]) in Scala. For instance, the following class **`Celsius`** defines a property **`degree`** which can be set only to values greater or equal than -273. [14]

```
class Celsius {
  private var d: Double = 0
  def degree: Double = d
  def degree_=(x: Double): unit = if (x >= -273) d = x
}
```

Now **`degree`** can be used as if it was declared as a variable.

```
val c = new Celsius;
c.degree = c.degree - 1;
```

3.1.5. Functions and Closure

1. A closure is a "code block" or a method pointer that is executed at a later point. Scala allows closures as shown in the following example where **`foo`** is a closure.

   ```
   val div = {(x:Double, y:Double) => println(x/y) }
   div(22,7);
   ```

2. Scala allows anonymous, curried, nested and higher order functions.

   ```
   //define anonymous function and call it
   println(((x:Double, y:Double) => x / y)(22,7));

   //nested inner functions can access outer's locals and arguments
   def outerFunction(str: String) = {
     def innerFunction() = {
       println(str);
     }
     innerFunction();
   }
   ```

3. Functions can be used as arguments to functions. For example a function **`forall`** is defined to be true if the predicate passed holds true for all elements of the passed array. Then the expression **`forall(row, 0 ==))`** tests whether **`row`** consists only of zeros. Here, the **`==`** method of the number 0 is passed as argument. This illustrates that methods can themselves be used as values in Scala. [14]

4. Functions are objects with **`apply`** methods so the function call **`foo(x)`** is actually **`foo.apply(x)`**. Special syntax exists for function applications appearing on the left-hand side of an assignment; these are interpreted as applications of an **`update`**

method. So, `a(i)` being the array element at index `i`, the assignment `a(i) = a(i) + 1` is interpreted as `a.update(i, a.apply(i) + 1)`. [14]

3.1.6. Case Classes and Pattern Matching

Case classes are regular classes which export their constructor parameters and provide a recursive decomposition mechanism via pattern matching. The Scala Tour gives a really nice example of representing untyped lambda calculus using case classes. It is very similar to the way I have used case classes to implement bind with expressions in Scala as discussed in Section 4.1.1.4. The example creates an abstract super class `Term` and three concrete case classes `Var`, `Fun`, and `App`.

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
```

Construction of case class instances does not require the `new` keyword. So for the above example we can create a `Term` by the following syntax.

```
Fun("x", Fun("y", App(Var("x"), Var("y"))))
```

Case classes implicitly come with methods like `equals`, which implements structural equality, and `toString` which prints the entire expression tree based on the way it was constructed as shown in the example below. The `==` operator calls the `equals` method.

```
val x1 = Fun("x", Fun("y", App(Var("x"), Var("y"))))
val y1 = Var("y")
val y2 = Var("y")
println(x1)         //prints Fun(x,Fun(y,App(Var(x),Var(y))))
println(y1 == y2)   //prints true
```

Case classes allow the use of pattern matching to decompose data structures. The following object defines a pretty printer function for the lambda calculus representation.

```
object TermTest extends Application {
  def print(term: Term): Unit = term match {
     case Var(n) =>
        Console.print(n)
     case Fun(x, b) =>
        Console.print("^" + x + ".") print(b)
     case App(f, v) =>
        Console.print("(") print(f)
        Console.print(" ") print(v)
        Console.print(")")
  }
}
```

The function **print** is expressed as a pattern matching statement starting with the **match** keyword and consisting of sequences of **case Pattern => Body** clauses similar to Java's switch statement.

3.1.7. Views

The Scala Overview [14] defines views as implicit conversions between types used to add new functionality to an existing type. It gives the following example of a Generic list to Set conversion. Notice that the Set is declared as a **trait**. Traits are similar to interfaces in Java, except they can have partial implementation.

```
abstract class GenList[T] {
  def isEmpty: boolean
  def head: T
  def tail: GenList[T]
  def prepend[S >: T](x: S): GenList[S]
}

trait Set[T] {
  def include(x: T): Set[T]
  def contains(x: T): boolean
}
```

A view from class **GenList** to class **Set** is introduced by the following method definition.

```
implicit def listToSet[T](xs: GenList[T]): Set[T] =
    new Set[T] {
      def include(x: T): Set[T] = xs prepend x
      def contains(x: T): boolean =
                  !xs.isEmpty && (xs.head == x || (xs.tail contains x))
    }
```

Hence, if **xs** is a **GenList[T]**, then **listToSet(xs)** would return a **Set[T]** . The only difference with respect to a normal method definition is the **implicit** modifier. This modifier causes them to be inserted automatically as implicit conversions. Say **e** is an expression of type **T**. A view is implicitly applied to **e** in one of two possible situations:

1. when the expected type of **e** is not (a supertype of) **T**, or
2. when a member selected from **e** is not a member of **T**.

For instance, assume a value **xs** of type **GenList[T]** which is used in the following two lines.

```
val s: Set[T] = xs;
xs contains x
```

The compiler would insert applications of the view defined above into these lines as follows:

```
val s: Set[T] = listToSet(xs);
listToSet(xs) contains x
```

3.2. Groovy

Groovy is a dynamic language for the Java Virtual machine and allows integration with Java objects and libraries. It provides many meta-programming features that can be useful for creating DSLs. This report discusses only those features of Groovy that I have used in my DSL creation.

Groovy supports metaobject protocol (MOP). MOP is an interpreter of the semantics of a program that is open and extensible. It determines what a program means and what its behavior is, and it is extensible in that a programmer can alter program behavior by extending parts of the MOP. The MOP may manifest as a set of classes and methods that allow a program to inspect the state of the supporting system and alter its behavior.[3]

Groovy is currently an evolving language. Many of the dynamic features have been added to the language very recently and are only available under the developer version of the language. It has sparse documentation available and most of the information on its feature descriptions below, has been taken from its online User Guide. [10]

3.2.1. Parentheses-less Methods and Named Parameters

Method calls in Groovy can omit the parentheses if there is at least one parameter and no ambiguity. For example, the statement **`println("Hello world")`** can be written as **`println "Hello world`**".

It is also possible to omit parentheses when using named arguments which makes for nicer DSLs.

```
search key: k, tree: bTree
pacman.move from: [1,4], to: [1,10]
```

3.2.2. Closure

Groovy provides support for closures. As an example lets look at defining action listeners inline through closures.

```
//Add a property Change Listener to myProperty
myProperty.propertyChange = { e -> println "property changed";}
```

The closure is passed to the method on the listener interface (**`propertyChange`**), and not the interface itself (**`PropertyChangeListener`**).

In Groovy, if a closure takes a single argument, it can be omitted in the definition and simple referred to as **`it`**.

```
def clos = { println it }
clos( "hello world" )      //prints "hello world"
```

In Groovy, when defining closures, **`this`** refers to the enclosing class, **`delegate`** refers to the enclosing object, which is either the same as **`this`** or a surrounding closure, and can be changed.

3.2.3. Categories

Categories are used to add new methods and properties to an existing class. To add a method to a class **`T`** we define a new class. Then we add a static method to it whose first parameter is of type **`T`**. To use this method we have to pass this class to the **`use`** keyword. Here is a simple example from the Groovy User Guide. [10]

```
class StringCategory {
    static String lower(String string) {
        return string.toLowerCase()
    }
}

use (StringCategory) {
    print "TeSt".lower() //prints "test"
}
```

Listing 3-1 Categories demo for adding methods to String class and its usage

3.2.4. DelegatingMetaClass

Each Groovy object has a meta class which intercepts calls to the object and can be used to add behavior to the object. It is possible to replace a meta class to adjust the default behavior of a class using **`DelegatingMetaClass`**. One way to replace a meta class using **`DelegatingMetaClass`** is the package name convention solution and this is the solution I have used in my DSL. A description of the how to use this solution to replace a meta class is given below.

Package Name Convention Solution
Any class can have a custom meta class loaded at startup time by placing the meta class into a well known package with a well known name.

```
groovy.runtime.metaclass.[YOURPACKAGE].[YOURCLASS]MetaClass
```

The following example from the Groovy user guide [10] shows how we can change the behavior of the String class. First we define the custom meta class.

```
package groovy.runtime.metaclass.java.lang

class StringMetaClass extends groovy.lang.DelegatingMetaClass{
  StringMetaClass(MetaClass delegate){
    super(delegate); //delegate contains the String instance
  }

  public Object invokeMethod(Object obj,
                             String methodName,
```

```
                                Object[] args){
    return "changed ${super.invokeMethod(obj, methodName, args)}"
  }
}

class DelegatingMetaClassPackageImpliedTest extends GroovyTestCase{
  void testReplaceMetaClass(){
    assertEquals "changed hello world", "hello world".toString()
  }
}
```

Listing 3-2 Modifying a `String` class method using `DelegatingMetaClass`

Notice that the actual code to use the enhanced features is very simple. There are no extra imports or any mention of the meta class. The mere package and name of the class tells the groovy runtime to use the custom meta class.

3.2.5. ExpandoMetaClass

The other way to augment an object's meta class in Groovy is by using the **`ExpandoMetaClass`**. It can dynamically add properties, methods, constructors and even static methods etc. to existing classes. Every class in Groovy has a **`metaClass`** property that can be used to get the **`ExpandoMetaClass`** instance. An example from the Groovy user guide [10] is given below which shows how to augment the behavior of the String class and add a method called **`swapCase`** to it.

```
String.metaClass.swapCase = {->
  def sb = new StringBuffer()
    delegate.each {
      sb << (Character.isUpperCase(it as char) ?
             Character.toLowerCase(it as char) :
             Character.toUpperCase(it as char))
    }
    sb.toString()
}
```

Listing 3-3 Adding new methods to `String` class using `ExpandoMetaClass`

To add a static method **`foo`** to the class **`Object`** we would use the syntax **`Object.metaClass.'static'.foo << {...}`**. To add a constructor to the **`String`** class we would say **`String.metaClass.constructor << {...}`** and so on.

By default the functionality added to the **`metaClass`** of a parent class does not get inherited by the derived classes when using **`ExpandoMetaClass`**. To enable this inheritance we need to call **`ExpandoMetaClass.enableGlobally()`** before the application starts, such as in the main method or servlet bootstrap. [10]

4. Rich Graphical User Interface DSL

To evaluate the ability of Scala and Groovy to be DSL hosts, I created a small JavaFX like DSL for rich graphical user interface (GUI) creation. The features of JavaFX roughly fall into the following 3 categories:

1. *Manipulating and querying arrays:*
   These are already well handled in Scala by features such as sequence comprehensions [5]. In Groovy, DSLs already exist to provide this kind of functionality. One such sample DSL is Quaere. [19]
2. *Reactive programming:*
   This includes features of JavaFX such as bind, triggers and inverse attributes. I am covering bind and inverse attributes in my DSL under incremental dependency-based evaluation. Triggers are a replacement of Java's class constructors and attribute setters and therefore the trigger functionality is already available in my host languages.
3. *control structures for threading:*
   Thread utility features will be provided by in my DSL.

Based on the above categories, my DSL provides features that facilitate GUI development such as dynamic updates of properties based on other properties, thread utilities and an operator that aids in creating animations.

4.1. DSL in Scala

The DSL created in Scala was accomplished using the Implicit conversion (views) feature of Scala. For details on Scala views please consult Section 3.1.7.

4.1.1. Incremental Dependency-Based Evaluation (Bind)

The **`bind`** feature of the DSL is similar to JavaFX **`bind`** discussed in Section 2.1.1

4.1.1.1. Bound Property

I created a new type of property called **`BoundProp`**. It is a generic class and can be declared with any type. It provides the ability to update a property automatically based on the value of another property. To understand how bound properties can be used let us first consider a trivial example.

```
import boundUtilities.BoundProp;

class Person(str: String){   //class declaration and constructor
    var name = new BoundProp[String];
    name := str;
}

object BindTester {
```

```
    def main(args: Array[String]) : Unit = {

        val user = new Person("Voltaire");
        var name = new BoundProp[String];

        //bi-directional binding between user.name and name
        name bind user.name;
        println(name);   //prints "Voltaire"

        user.name := "Homer";
        println(name);   //prints "Homer"
    }
}
```

Listing 4-1 Bi-directional binding of two properties

In the above example the variable **name** is bi-directionally bound to **user.name** and one gets automatically updated whenever the value of the other changes.

A property change event needs to be fired when the value of a **BoundProp** is updated in order to notify other properties bound to it. For this reason I created the **:=** method that is used to assign a value to the **BoundProp**. Scala does not allow overloading the **=** operator, so I had to use a different assignment operator. The **:=** operator sets the new value of the **BoundProp** and also fires the property change notifications. The above example uses the **:=** operator to assign the new value to **user.name**. Given below is an excerpt from the **BoundProp** class that shows the implementation of bind.

```
class BoundProp[T] extends PropertyChangeSupport with
PropertyChangeListener {

  var value: T = _;

  //bi-directional binding done between “this” and “other”
  def bind(other: BoundProp[T]){
      this.addPropertyChangeListener(other);
      other.addPropertyChangeListener(this);
      if(other.value != null){
        this := other.value;
      }
  }

  //update "this" on change notification from a property bound to it
 def propertyChange(evt: PropertyChangeEvent){
    //using "this :=" rather than "this =" so that its property
    //change event is fired notifying other properties bound to it
    if(!(evt.getNewValue().asInstanceOf[T]).equals(value)){
          this := evt.getNewValue().asInstanceOf[T];
      }
  }

  //overloaded operator for assignment with RHS value of type T
  def :=(newValue : T): BoundProp[T] = {
   val oldValue = value;
   value = newValue;
   firePropertyChange("value", oldValue, newValue);
```

```
   this;
  }

  //overloaded assignment operator with RHS of type BoundProp[T]
  def :=(newProperty : BoundProp[T]): BoundProp[T] = {
    val oldValue = value;
    value = newProperty.value;
    firePropertyChange("value", oldValue, value);
    this;
  }
  ...
}
```

Listing 4-2 Bind implementation in `BoundProp` class

4.1.1.2. Bound Swing Components

In the last section I considered a trivial case (Listing 4-1) to show the usage and working of the **`bind`** operator on properties. The real value of **`bind`** operator can be seen in the context of the model-view-controller pattern. Here the **`bind`** operator is used to bind the *view* (user interface) of an application to the *model* (data) of the application. Consider the following example where a model's **`author`** attribute is bound to a Swing component.

```
import boundUtilities.BoundJComboBox._;

class Author(n: String) {
     var name = n;
     var books: String = _;
}

class Model{
  var author = new BoundProp[String];
}

object View {
  def main(args: Array[String]) : Unit = {
    ...
    val model = new Model;
    val author1 = new Author("Jane Austen");
    val author2 = new Author("Charlotte Bronte");
    val authors: Array[Object] = Array(author1.name, author2.name);

    val authorList = new JComboBox(authors);

    //Implicitly converting model.author from BoundProp[String]
    //to BoundProp[Object]
    authorList.selectedItem bind model.author;
    model.author := "Charlotte Bronte";
    ...
}
```

Listing 4-3 Binding of model's attribute to view's component

In the example the view is automatically updated on changes in the model's data. The **`selectedItem`** property of the **`authorList`** is bound to the **`author`** attribute of the model

class using the **bind** operator. This creates a bi-directional binding between the two and change in one is reflected immediately in the other. It eliminates the boilerplate of writing listeners for notification of updates in Swing components and the corresponding code to update the attributes of the model. Notice the special syntax of **import** statement at the top, this is the statement that makes available my DSL's **bind** feature to the user. More detail of how **bind** is implemented in Swing components and how this special import statement works is given at the end of this section.

The rich GUI DSL also allows binding between two Swing components. I can take the above example and extend it to include binding between the **authorList** combo box and a list.

```
import boundUtilities.BoundJComboBox._;
import boundUtilities.BoundJList._;

class Author(n: String) {
     var name = n;
     var books: String = _;
}

class Model{
  var author = new BoundProp[String];
}

object BindTester {
   def main(args: Array[String]) : Unit = {
    ...
     val model = new Model;

     val author1 = new Author("Jane Austen");
     author1.books = "Pride and Prejudice, Emma, Persuasion";

     val author2 = new Author("Charlotte Bronte");
     author2.books = "Jane Eyre, Shirley, Villette, The Professor";

     val authors: Array[Object] = Array(author1.name, author2.name);
     val books: Array[Object] = Array(author1.books, author2.books);

     val authorList = new JComboBox(authors);
     val booksList = new JList(books);

     authorList.selectedItem bind model.author;
     authorList.selectedIndex bind booksList.selectedIndex;

     model.author := "Charlotte Bronte";
     ...
}
```

Listing 4-4 Binding of two `Swing` components

When the **author** is selected from the combo box, the selected item in the **booksList** updates to show the books by that author. This binding is bi-directional. Therefore, changing books in the list selection will update the author name in the combo box as well. Below is a screenshot of the demo application.

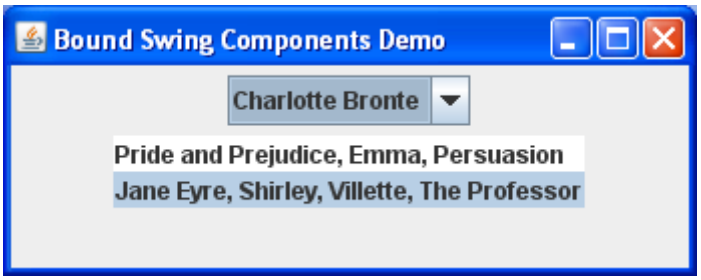

Figure 2. Screenshot of `Swing` component binding demo

The **`bind`** feature is added to existing Swing components by creating extension classes that contain the added functionality. JavaFX also, has its custom UI classes created on top of Swing components to provide extra functionality. The code for my **`BoundJTextField`**, an extension class to **`JTextField`** is given below.

```
package boundUtilities;

import java.beans._;
import javax.swing.JTextField;
import java.awt.event._;

class BoundJTextField(textField: JTextField){
  var text = internalBind(); //text internally bound to textField's
                             //text property

  //bi-directionally bind textfield's text to local variable text
  private def internalBind(): BoundProp[String] = {
    var temp = new BoundProp[String];

    //Update text whenever the textField's text property changes
    textField.addActionListener(new ActionListener{
      def actionPerformed(e:ActionEvent): Unit = {
        temp := textField.getText();
      }
    });

    //Update textField's text property when local text changes
    temp.addPropertyChangeListener(new PropertyChangeListener{
      def propertyChange(evt: PropertyChangeEvent){
  if(!textField.getText().equals(evt.getNewValue())){
    textField.setText(evt.getNewValue().asInstanceOf[String]);
        }
      }
    });
    return temp;
  }
}
//companion object
object BoundJTextField {
  implicit def JTextField2BoundJTextField(tf: JTextField) =
                                                  new BoundJTextField(tf)
}
```

Listing 4-5 Bind implementation for `JTextField's` companion class

The class **BoundJTextField** has an attribute **text** of type **BoundProp[String]**. This is the attribute that users bind to when they call bind on a **JTextField's** text property. Internally, it takes care of bi-directional updates between the **JTextField's** text property and itself. To use the views feature of Scala, I created a companion object **BoundJTextField** with the same name as my class, as shown above, and declared a method for implicit conversion from **JTextField** to my **BoundJTextField**. Now I can use the special import syntax with the "_" to allow implicit conversion of **JTextField** to **BoundJTextField**. The user can now use the bind feature as if it was a part of **JTextField**, as shown below.

```
object BindTester {
  def main(args: Array[String]) : Unit = {

    import boundUtilities.BoundJTextField._;
    {
      val field1 = new JTextField;
      val field2 = new JTextField;

      //implicitly converting field1 and field2 to BoundJTextField
      field1.text bind field2.text;
    }
  }
}
```

Listing 4-6 Binding of two text fields in the DSL in Scala

The above example also shows how we can limit the scope of the implicit conversion by adding the import statement only around the code that needs it. For details on views and companion objects in Scala, see section 3.1.7.

4.1.1.3. Bound Bean Property (An Alternate Approach to bind)

In the above approaches, I created custom extension classes for every class whose properties were to support bind. Another approach that I tried for adding bind functionality was by using bound properties of JavaBeans. This was to be a generic approach to create a bound property from any property of any class.

Beans are reusable components that follow strict naming conventions. For example, properties need to have their accessor and mutator methods begin with **get** and **set** respectively. Based on these naming conventions, a **java.beans.Introspector** can create a **java.beans.BeanInfo** object for a given class. Then we can use reflection to search for features, such as properties and methods, provided by the class. If the class has bound properties, they can be used to notify interested listeners of updates to the property value. A bound property must implement two mechanisms: [13]

1. Whenever the value of the property changes, the bean must send a **PropertyChange** event to all registered listeners. This change can occur when the **set** method is called or when the user interface changes the value of the property.

2. To enable interested listeners to register themselves, the bean has to implement the following two methods:

```
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

I created a **BoundBeanProp** class that accepts an Object and the attribute of that object that is to be bound. Then I extract the property descriptor for that field using Introspection and Reflection. In the bind method of the property, I extract the **eventSetDescriptor** for the **propertyChange** event, using which I add listeners that would notify me when value of the property (corresponding to the above field) change. Consider the following example where we want to bi-directionally update the value of a **JTextField's** text property with **modelData**.

```
val field1 = new JTextField;
var modelData = new BoundProp[String];
var field1Text = new BoundBeanProp[String](field1, "text");

//bi-directional binding between textfield's text property & modelData
field1Text bind modelData;
field1.setText("Duke");  //value of modelData updated to "Duke"
```

Listing 4-7 Sample code for BoundBeanProp's bind usage

In the bind method of the **BoundBeanProp** class the specified field's listeners are added to the other property and vice versa. So in the above example the call to bind adds two listeners. One, on **field1.text** to update the **modelData** value and the other on the **modelData** to update the value of **field1.text**. Since I had a **propertyDescriptor** to the selected field (**text** in my example), I could read and write to the field by using **getReadMethod** and **getWriteMethod** on the **propertyDescriptor** and invoking the resulting methods.

The problem I encountered with the above approach was that for most Swing components, such as the **JTextField** component from the above example, property change events were not fired properly. So, in the above example, changing the value of **field1.text** property by calling the **setText** method fires a property change event (updating the **modelData**), but changing its value from the UI does not fire any property change events and hence has no effect on **modelData**.

The above problem shows one reason why having native bound properties in Java would have been a good idea. They would ensure that developers of classes like the Swing components would not have to manually keep track of and fire property change events, for all possible scenarios of updates to the component properties. Work is being done to alleviate this issue in Java. For reference consult [20] and [21].

One approach to solving the above problem in **BoundBeanProp** was to create custom extension classes from Swing components and adding fire property change events on updates. A sample implementation of a custom class for above example is given here.

```
class TextField extends JTextField {
  class TextFieldDocumentListener extends DocumentListener {
    def insertUpdate(e : DocumentEvent) : Unit = {
      firePropertyChange("text", null, getText());
    }
    def removeUpdate(e : DocumentEvent) : Unit = {
      firePropertyChange("text", null, getText());
    }
  }
  getDocument().addDocumentListener(new TextFieldDocumentListener());
}
```

Listing 4-8 Custom `TextField` class for use with `BoundBeanProp`

However, using `DocumentListener's` update methods resulted in notifications being fired on every key press in the text field. This makes the resulting code slower. Besides, creating a custom class for each component beats the purpose of using bound properties of JavaBeans, which was to create a single generic class to add the bind functionality.

4.1.1.4. Bind with Expressions

Many times we need a property whose value depends not directly on other properties but an expression involving those properties. To provide this facility in my DSL, I created the bind with expressions feature. Consider the following sample application to clarify the use of this feature.

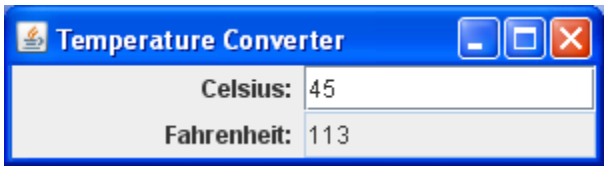

Figure 3. Screenshot of Bind with Expressions demo application

The demo application provides the simple functionality of converting temperature from degree Celsius to degree Fahrenheit. The user enters a temperature in the Celsius text box and the Fahrenheit text box is automatically updated to show the corresponding temperature in degree Fahrenheit. The following is the application code segment that shows how the value of Fahrenheit textbox gets updated.

```
import utils.Term._;
import boundUtilities.BoundPropDouble;
import boundUtilities.BoundJFormattedTextField._;

object temperatureConverter {
  def main(args: Array[String]) : Unit = {
    ...
    val celsiusField = new JFormattedTextField;
    val fahrenheitField = new JFormattedTextField;
    fahrenheitField.setEditable(false);

    var Celsius = new BoundPropDouble;
```

```
    var Fahrenheit = new BoundPropDouble;
    Fahrenheit bind (Celsius * 1.8) + 32.0;    //unidirectional bind

    Celsius := 45;

    celsiusField.value bind Celsius;
    fahrenheitField.value bind Fahrenheit;
    ...
  }
}
```

Listing 4-9 Code excerpt from bind with expressions demo application

Here we have two **`JFormattedTextField`** objects which following the model-view Pattern are bound to the **`Celsius`** and **`Fahrenheit`** properties respectively. The bind on these fields is provided by implementing a **`BoundJFormattedTextField`** class in the same way as other Swing Components in Section 4.1.1.2. What is different here is that **`Fahrenheit`** property, besides being bound to a field, is also bound to an expression containing another property. Notice that bind with expressions is a unidirectional bind. This is because the value of properties in an expression cannot always be uniquely determined when the value of the left hand side changes. For example, in a case like **`sum bind number1 + number2`** the value of **`number1`** and **`number2`** cannot be uniquely resolved when the value of **`sum`** changes.

To enable bind with expressions, I needed to store the entire unevaluated expression so that every time the properties in the expression got updated I could recalculate the value of the expression and update the value of the left hand side. In order to do this, I created a **`Term`** class which can store and evaluate an expression tree. So, in the above example the expression **`(Celsius * 1.8) + 32.0`** is implicitly converted to an object of type **`Term`** as shown in `Figure 4. Expression tree for (Celsius * 1.8) + 32.0.`

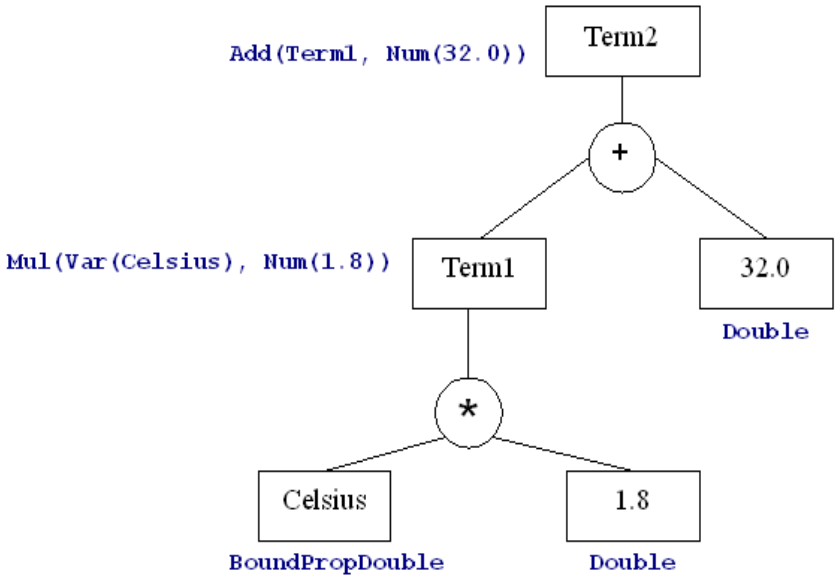

Figure 4. Expression tree for `(Celsius * 1.8) + 32.0`

The **Term** class consists of case classes **Num**, **Var**, **Add** and **Mul**. These represent terms of type number (e.g. **34**, **6** etc.), variables (e.g. **x**, **num** etc.), addition of two terms (e.g. **x+3**, **a+b**, **2+5** etc.) and multiplication of two terms (e.g. **num*2**, **3*6**, **a*b** etc.) respectively. For evaluation of the **Term** I use pattern matching. For more information on Scala's case classes and pattern matching refer to Section 3.1.7. The **Term** class overloads **+** and ***** operators so that addition or multiplication of two Terms results in a **Term**.

If you look at above example (Listing 4-8) you will notice that I use **BoundPropDouble** instead of the generic **BoundProp** class with type **Double**. The reason is that, properties declared as **BoundProp** (with any type) will not allow + and * operations on them since there is no way to ensure that all types with which **BoundProp** can be declared will always provide **+** and ***** methods.

The excerpt from the **Term** class implementation is given below. Also shown are the definitions of the case classes and the implicit conversion.

```
abstract class Term extends PropertyChangeListener {
  var bProperty = new BoundPropDouble;

  def +(r:Term) : Term = {
    var myTerm: Term = new Add(this, r);

    this.bProperty.addPropertyChangeListener(myTerm);
    r.bProperty.addPropertyChangeListener(myTerm);

    return myTerm;
  }

  def propertyChange(evt: PropertyChangeEvent){
    bProperty := eval;
  }

  def eval(): Double = this match {
     case Num(m) =>
        m;
     case Var(m) =>
        m.value;
     case Add(l: Term, r: Term) =>
        bProperty := l.eval + r.eval;
        bProperty.value;
     case Mul(l: Term, r: Term) =>
        bProperty := l.eval * r.eval;
        bProperty.value;
  }
  ...
}

object Term {
  implicit def Double2Term(d : Double) = new Num(d);
}

case class Num(n: Double) extends Term{
  bProperty := n;
```

```
}

case class Var(v: BoundPropDouble) extends Term{
  bProperty := v;
}

case class Add(l: Term, r: Term) extends Term{
  bProperty := eval;
}

case class Mul(l: Term, r: Term) extends Term{
  bProperty := eval;
}
```

Listing 4-10 Excerpt from `Term` class implementation for bind with expressions

If you look at the code for the **+** method you will notice that each **`Term`** of type **`Add`** is a listener to updates in its left and right values. In an expression like **`a + (b + c), b + c`** evaluates to **`Term`** of type **`Add`** and this **`Term`** has property change listeners attached to both **`b`** and **`c`**. Then, **`a`** and this **`Add`** type **`Term`** combine to form another **`Term`** which has listeners attached to **`a`** and the previous **`Term (b + c)`**. So, a change in the value of **`b`** triggers the **`PropertyChange`** in **`Term (b + c)`** and it value gets re-evaluated. This change in value of **`Term (b + c)`** in turn triggers a change in **`Term a + (b + c)`** and that also gets re-evaluated. This is how property change events propagate up in the expression tree.

The **`bind`** to **`Term`** was implemented along the same pattern as **`bind`** to Swing components where the binding is actually done to a local variable (**`bProperty`**) which is internally bound to the final value of the expression.

4.1.2. Animate Feature

For my DSL, I created a feature to facilitate animation which mimicks the functionality provided by the JavaFX **`dur`** operator discussed in Section 2.1.2. The animate feature is available on bound properties. It accepts an array of values and a time parameter in milliseconds. Based on these two parameters it updates the value of the property to the array of values over the span of time. The animate feature creates a background thread which assigns the range of values from the array into the bound property. To show the power of the animate feature lets consider a small animation demo. The demo uses the animate feature to update the panel background color from yellow to blue to red between a 3 second interval.

Figure 5 Screenshot of Animation demo at 1 second and 3 second

The code to animate the panel's background color is given below. Notice how just a single statement handles the entire animation task.

```
import boundUtilities.BoundProp;
import boundUtilities.BoundJPanel._;

object animationTester {
  def main(args: Array[String]) : Unit = {
    ...
    val colors = Array(Color.YELLOW, Color.BLUE, Color.RED)

    val panel = new JPanel
    panel.backGround animate (colors, 3000)
    ...
  }
}
```

Listing 4-11 Code excerpt from animation demo

The panel is implicitly converted to a **`JBoundPanel`** which is implemented the same way as other Bound Swing Components described in Section 4.1.1.2. This implicit conversion allows features of my DSL, like bind and animate, to be called on the **`JPanel`** objects. The implementation of **`animate`** is given below. The first parameter of is of type **`Seq`** which is the parent class of all sequence type classes in Scala. This enables the animate feature to accept all types of sequence objects such as **`Array`**, **`List`**, **`Range`** etc.

```
class BoundProp[T] extends PropertyChangeSupport with
PropertyChangeListener {
  ...
  def animate(seq: Seq[T], millisecs: Long) = {
    val internal = this;
    var t = new Thread(new Runnable(){
          def run = {
              for (i <- 0 to seq.length-1) {
                internal := seq(i);
                Thread.sleep(millisecs/(seq.length));
              }
            }
        });
    t.start();
  }
}
```

Listing 4-12 Implementation of `animate` feature in the `BoundProp` class

Scala's **`Range`** class provides some nice syntactic sugar for the animate feature. It lets you specify the start value, end value and a step value for the sequence and generates the entire sequence. So instead of writing

```
countDown animate (Array(10,9,8,7,6,5,4,3,2,1,0), 10000);
```

we can use the much nicer **`Range`** syntax to do animation in the DSL.

```
countDown animate (10 to 0 by -1, 10000);
```

4.1.3. `doOutside` and `doLater`

The next set of features I implemented for my DSL were related to concurrency which is important when creating rich GUIs. The **`doOutside`** feature provides the same functionality as JavaFX's **`do`** statement discussed in Section 2.1.3. It allows you to execute a portion of code in a separate thread outside of the current thread. To illustrate its use, let us consider how we could rewrite the **`animate`** code (Listing 4-11) given above in Section 4.1.2.

```
def animate(seq: Seq[T], millisecs: Long) = {
  val internal = this;
  doOutside{
    for (i <- 0 to arr.length-1) {
      internal := arr(i);
      Thread.sleep(millisecs/(seq.length));
    }
  }
}
```

Listing 4-13 Usage of `doOutside` in `animate` feature implementation

We can see how the **`doOutside`** feature allowed us to remove the boiler plate code of creating a new thread, creating its inner class and starting the thread. So, we can just write the code that we want to execute in the new thread and let the DSL worry about all the inner details of thread creation and running.

The **`doLater`** feature provides the same functionality as JavaFX's do later statement discussed in Section 2.1.3 or the **`java.awt.EventQueue.invokeLater`**. You can use it to asynchronously execute a section of your code in the EDT. Lets consider a small example, inspired from Core Java [12], which creates a new thread and also uses **`java.awt.EventQueue.invokeLater.`** Then we will look at the same code using **`doOutside`** and **`doLater`** features of the DSL. The demo just creates a **`JComboBox`** and keeps updating the items in it, by either adding a new random value or deleting an existing value.

```
object ThreadTester {
  def main(args: Array[String]): Unit = {
    ...
    val combo = new JComboBox();
    var t = new Thread(new Runnable(){
         def run = {
           var generator = new Random();
           while (true){
               EventQueue.invokeLater(new Runnable(){
                     def run(){
                       var i = Math.abs(generator.nextInt());
                       if (i % 2 == 0) combo.insertItemAt(i, 0);
                       else if (combo.getItemCount() > 0)
                         combo.removeItemAt(i % combo.getItemCount());
                     }
```

```
                    });
                Thread.sleep(10000);
            }
        }
    });
    ...
  }
}
```

Listing 4-14 Original code from `ThreadTester` application

We can rewrite the above code in a much more succinct manner by using the **`doOutside`** and **`doLater`** features of the DSL as shown below.

```
import boundUtilities.ExtendedObject._;
object ThreadTester {
   def main(args: Array[String]): Unit = {
     ...
     val combo = new JComboBox();
     doOutside{
       var generator = new Random();
       while (true){
         doLater{
           var i = Math.abs(generator.nextInt());
           if (i % 2 == 0) combo.insertItemAt(i, 0);
           else if (combo.getItemCount() > 0)
             combo.removeItemAt(i % combo.getItemCount());
         }
         Thread.sleep(10000);
       }
     }
     ...
   }
}
```

Listing 4-15 Rewrite of `ThreadTester` using DSL features of `doOutside` and `doLater`

The implementation of the **`doLater`** and **`doOutside`** features of the DSL is given below. They are methods in an extension to **`Object`** class and take closures as parameters, executing them asynchronously or synchronously. To use these methods, add **`import boundUtilities.ExtendedObject._;`** statement to the code.

```
import java.awt.EventQueue;

object ExtendedObject {
  def doLater(func: =>Unit): Unit = {
        EventQueue.invokeLater(new Runnable(){
                def run = {
                  func
                }
            });
  }

  def doOutside (func: =>Unit): Unit = {
    var t = new Thread(new Runnable(){
                def run = {
```

```
                    func
                }
            });
        t.start();
    }
}
```

Listing 4-16 Implementation of `doLater` and `doOutside` features

4.2. DSL in Groovy

To evaluate Groovy's dynamic features I created the same rich graphical user interface DSL in Groovy as I had done in Scala. I used Groovy's meta class augmenting capabilities to implement the new features.

4.2.1. Incremental Dependency-Based Evaluation (Bind)

One of the features implemented in the DSL was the **`bind`** feature which resembles the JavaFX **`bind`** feature. For details on the working of **`bind`** please consult Section 2.1.1.

4.2.1.1. Bound Property

I created a **`BoundProp`** class whose objects can use **`the`** bind feature to automatically update their value based on other **`BoundProp`** objects. It is a generic class that can be initialized to any type. The following is a trivial example that shows how the **`bind`** feature is used to bi-directionally **`bind`** two String type properties. The class **`Book`** has a **`BoundProp title`** of type String and we are binding **`title`** property of **`book1`** to **`title`** property of **`book2`**.

```
import bComponents.BoundProp;

class Book {
  BoundProp<String> title = new BoundProp<String>(this);
}

class BindTester {
  static void main(args) {
    Book book1 = new Book();
    Book book2 = new Book();
    book1.title.bind book2.title;

    book1.title.set("Alice in Wonderland");
    assert (book2.title.value == "Alice in Wonderland");
}
```

Listing 4-17 Bidirectional binding of two `String` type `BoundProp` objects

Groovy supports invocation of methods without parentheses when we only have a single parameter, so we can call the **`bind`** method on **`book1.title`** without the parentheses. Now when the **`set`** is called for **`book1.title`**, the **`title`** property of **`book2`** automatically gets updated to reflect the change. The implementation of **`bind`** in the **`BoundProp`** class is given below.

```
package bComponents

import java.beans.*;

class BoundProp<T> extends PropertyChangeSupport implements
PropertyChangeListener {
  T value;

  def BoundProp(Object source){
    super(source);
    value = new T();
  }

  //bi-directional binding done between this and other
  def bind(BoundProp<T> other){
    this.addPropertyChangeListener(other);
    other.addPropertyChangeListener(this);
  }

  void propertyChange(PropertyChangeEvent evt){
    if(!evt.getNewValue().equals(value)){
      this.set(evt.getNewValue())
    }
  }

  BoundProp<T> set(T newValue){
    T oldValue = value;
    value = (T)newValue;    //hack to fix problem with Groovy generics
    firePropertyChange("value", oldValue, newValue);
    this;
  }
  ...
}
```

Listing 4-18 Implementation of bind in `BoundProp` class

The **`bind`** method assigns listeners between both the properties. On update of a property's value I needed to fire the property change event. To do this, I had to create an assignment mechanism that handled this. Groovy only allows operator overloading on a limited set of operators and assignment operator (**=**) is not one of them. Groovy does not allow operators as valid identifiers for method names so I could not use the Scala (**:=**) syntax either. Therefore I created the **`set`** method on **`BoundProp`**. When we use this method to change the value of a property, the other properties that directly or indirectly depend on it get notified and can update their values accordingly. The generics in Groovy are somewhat recent and error prone, therefore I had to include some hacks like type casting (**`value = (T)newValue;`**) in cases where for some reason the Groovy generics failed to act as expected.

4.2.1.2. Bound Swing Components

As mentioned in Section 4.1.1.2, the real use of bind is in case of applications based on the model-view-controller pattern, where the data from the backend (model) is used to populate the user interface (view). To keep the model and view in sync we can use

binding between model's properties and view's components. Consider the following example where two text fields are bi-directionally bound to a single property **`title`**. This means that the text field's are also indirectly bound to each other.

```
import javax.swing.*;
import bComponents.BoundProp;

class Model{
  BoundProp<String> title = new BoundProp<String>(this);
}

class BindTester {
  static void main(args) {
     ...
     JTextField field1 = new JTextField("Default");
     JTextField field2 = new JTextField("Default");

     Model model = new Model();

     field1.property.text.bind model.title;
     field2.property.text.bind model.title

     //Issue: ActionListener not getting invoked on setText
     //Does get invoked on UI input
     field1.text("Les Miserables");
     ...
  }
}
```

Listing 4-19 Indirect binding of two text field's

The screen shots from the sample application are given below.

Figure 6 Screenshots from model-view binding demo.

The first screen shows the application when it starts. We can see that although the value of **`field1`** was updated before the application starts, it has not effected the value of **`field2`**. The problem is that the **`setText`** method of **`JTextField`** does not fire a property change event. Once again, if Java had native bound properties built in to it, this would not have happened. The next screen shows the value of both text fields when **`field2`** is updated through the user interface. The updated value of **`field1`** confirms that the problem is only when the text field is set using the **`setText`** method. The automatic update works in all other scenarios like setting the value of **`model.title`**, or updates to either text field through the user interface.

The DSL also allows binding of two Swing components directly, so in the above example we could have also written the following, to bi-directionally bind the two text fields to each other.

```
field1.property.text.bind field2.property.text
```

First of all, note that we use the regular **JTextField** class. There are no extra imports required to provide bind functionality to it. The **bind** is added to the **JTextField** by using **DelegatingMetaClass** and the package name convention solution given in Section 3.2.4.
Every time any property is accessed on the **JTextField**, the **getProperty** method is intercepted by my **DelegatingMetaClass**. If the name of the property is "**property**" I return my class **BoundJTextField's** object but for all other properties the call is forwarded to the regular **JTextField** class. The **BoundJTextField** has a **BoundProp text** of type String. This property is internally bound to the **JTextField's text** property. Any component or property binding to the **JTextField** is actually bound to this local **text** property which in turn updates the **JTextField's text** property. The added **property** call on a field when binding, gives rise to more verbose syntax but is necessary to keep the normal operations of a **JTextField** intact. Otherwise, we could rig the **JTextField** to return **BoundJTextField's text** property on **field1.text** call. In this case the **bind** would work but many other normal operations which expect a String and not a **BoundProp<String>** would get affected. The implementation of my **DelegatingMetaClass** for **JTextField** and the **BoundJTextField** is given below.

```
package groovy.runtime.metaclass.javax.swing;

import groovy.lang.*;
import bComponents.BoundJTextField;

class JTextFieldMetaClass extends DelegatingMetaClass{

  JTextFieldMetaClass(MetaClass metaclass){
    super(metaclass);
  }

  public Object getProperty(Object object, String propName){
    //If "property" then return a BoundJTextField object
    if(propName == "property"){
      new BoundJTextField(object);
    }
    //for all other properties, forward to JTextField
    else{
      super.getProperty(object, propName);
    }
  }
}
```

Listing 4-20 The DelegatingMetaClass that intercepts the getProperty methods on JTextField

```
package bComponents

import javax.swing.*;
import java.awt.*;

class BoundJTextField {
  JTextField component;
  BoundProp<String> text;

  public BoundJTextField(JTextField c){
    component = c;
    text = new BoundProp<String>(this);
    internalBind();
  }

  BoundProp<String> getText(){ text; }

  void internalBind(){
    component.actionPerformed = { e -> text.set(component.getText()) };
    text.propertyChange = { e ->
              if(!component.getText().equals(e.getNewValue())){
                 component.setText(e.getNewValue());
              }
            };
  }
}
```

Listing 4-21 The `BoundJTextField` class that implements the bind feature

4.2.1.3. Bind with Expressions

For the bind with expressions feature, I did some research to try to come up with a solution better than what we ended up employing in Scala for the DSL, but at the time Groovy like Scala did not provide any feature that would automatically enable the handling of the unevaluated expression. For bind with expression we needed the unevaluated expression that could be reevaluated every time one of the properties that made up the expression changed. The aim was to find something like the macros available in LISP and Scheme but Groovy does not seem to have that capability. Since the Term class implementation that we did for the DSL in Scala did not employ any dynamic feature of the language to aid in its creation, I decided that redoing it in Groovy would not give me any advantage on evaluating the nature of Groovy as a DSL Host. I have therefore not implemented Bind with Expressions in Groovy.

In May 2008 the latest version of Groovy (1.6-beta-1) was released. It comes with a new feature called AST Transformations. According to the announcement by the Groovy creators on the mailing list:

"When the Groovy compiler compiles Groovy scripts and classes, at some point in the process, the source code will end up being represented in memory in the form of a Concrete Syntax Tree, then transformed into an Abstract Syntax Tree. The purpose of AST Transformations is to let developers hook into the compilation process to be able to modify the AST before it is turned into bytecode that will be run by the JVM."

This new feature seems promising for the implementation of my bind with expressions feature.

One currently available implementation of this AST Transformation is the **@Bindable** annotation that generates a **java.beans.PropertyChangeSupport** object, the corresponding methods to add listeners, and a setter that uses the property change support. This gives us a bound JavaBeans property without the boilerplate code.

4.2.2. Animate Feature

The next feature I implemented for my DSL was the **animate** feature which was built to facilitate the creation of animations. It was based on the JavaFX's **dur** operator which I discussed in Section 2.1.2. It takes an list of values and assigns them to a property over the specified amount of time. Below is an excerpt from a demo animation application which uses **animate**. The application animates the background color of a rectangle, changing it to a range of different colors over a period of 5 seconds.

```
import bComponents.BoundProp;
import javax.swing.*
import java.awt.*

class animationTester {
  static void main(args){
    ...
    def colors = [Color.YELLOW, Color.BLUE, Color.GREEN, Color.RED]
    JPanel panel = new JPanel();

    panel.property.background.animate(colors, 5000)
    ...
    }
}
```

Listing 4-22 Excerpt from demo application animating the background of a rectangle using the animate feature

Notice that we need a single line of code to animate the background color. We do not need any special import statements but we do need the more verbose **panel.property** syntax for the same reason as discussed in Section 4.2.1.2. The **panel** gets the **animate** in the same way as discussed for components in the previous Section on Bound Swing Components. The **panel** has a local property **background** which is of type **BoundProp<Color>**. It is internally bound to the **background** property of the **JPanel** class which is of type **Color**. The **animate** feature is then implemented just once inside the **BoundProp** class and is available to properties as well as Swing components. The implementation of **animate** in **BoundProp** is given below.

```
package bComponents

import java.beans.*;

class BoundProp<T> extends PropertyChangeSupport implements
PropertyChangeListener {
```

```
  ...
  def animate(Collection<T> coll, Long milliSecs){
    Closure func = {
       for (i in 0..coll.size-1) {
         set(coll[i]);   //set value in BoundProp
         int tempSleep = milliSecs/(coll.size);
         Thread.sleep(tempSleep);
       }
    };
    new Thread(func).start();
  }
}
```

Listing 4-23 Implementation of `animate` feature in `BoundProp` class

4.2.3. `doOutside` and `doLater`

Another pair of features I implemented in the DSL in Groovy was the thread related **`doOutside`** and **`doLater`**. They provided the same functionality as JavaFX's **`do`** and **`dolater`** discussed in Section 2.1.3 or the thread features in our DSL in Scala from Section 4.1.3.
I explored three different approaches to creating the **`doOutside`** and **`doLater`** features in Groovy.

The first approach was using Groovy's categories (Section 3.2.3). I created a **`ThreadUtilCategory`** class which has **`doOutside`** and **`doLater`** as static methods. To use the features we just need to put the **`ThreadUtilCategory`** class in the **`use`** statement as given below.

```
use(ThreadUtilCategory){
  doLater{
    println("This will run in the EDT");
  }
  doOutside{
    println("This will run in a new Thread");
  }
}
```

Listing 4-24 Usage of the `ThreadUtilCategory` in the DSL in Groovy

The advantage is that since we are not trying to extend the Object class this approach is more scalable. The features are only available within the scope of the **`use`** statement and do not affect anything outside. The implementation of the **`ThreadUtilCategory`** is given below.

```
import javax.swing.*;

class ThreadUtilCategory {
  static def doLater(Object obj, Closure func) {
    SwingUtilities.invokeLater(func);
  }
  static void doOutside(Object obj, Closure func){
    new Thread(func).start(); //Closure implements Runnable
  }
```

```
}
```

Listing 4-25 Implementation of the Thread Utility Category in Groovy

Notice that the **`Closure`** class in Groovy implements the **`Runnable`** interface and can therefore be passed directly in the **`Thread`** constructor.

The second approach I tried was using the **`DelegatingMetaClass`** of Groovy. The idea was to augment the meta class of the Object class and add the functionality there. The meta class intercepts all static methods on Object. If the name of the method matches the name of my features then the meta class executes the required thread procedures. All other static methods are passed on to the Object class for regular execution. The implementation of the meta class is given below.

```
import javax.swing.SwingUtilities;

class ObjectMetaClass extends DelegatingMetaClass{

  ObjectMetaClass(MetaClass metaclass){
    super(metaclass);
  }

  public Object invokeStaticMethod(Object a_object,
                                   String a_methodName,
                                   Object[] a_arguments){
    if(a_methodName == "doLater"){
      SwingUtilities.invokeLater((Closure)a_arguments[0]);
    }
    else if(a_methodName == "doOutside"){
      new Thread((Closure)a_arguments[0]).start();
    }
    else{
      super.invokeStaticMethod(a_object, a_methodName, a_arguments);
    }
  }
}
```

Listing 4-26 Implementation of the `DelegatingMetaClass` for `Object` with `doOutside` and `doLater` functionality

The usage of the meta class is shown below. Notice how we need to call the features on the Object class. This is because **`DelegatingMetaClass`** does not implement inheritance so the features are only available to the Object class.

```
Object.doLater{
  println("I am running in the EDT");
}
Object.doLater{
  println("I am running in new Thread");
}
```

Listing 4-27 Usage syntax of `doOutside` and `doLater` implemented in `Object` meta class

The third approach I tried was to use the **`ExpandoMetaClass`** (Section 3.2.5) since it allows the added functionality to be inherited by using its **`enableGlobally()`** method.

```
Object.metaClass.'static'.doLater << {SwingUtilities.invokeLater(it);}
ExpandoMetaClass.enableGlobally();

doLater {
  println("I should run in the EDT ");
}
```

Listing 4-28 `doLater` with `ExpandoMetaClass`

But the above code does not work and it still requires a call to **`Object.doLater`** to run the method. It does not work with **`String.doLater`** or **`this.doLater`** syntax but surprisingly works for **`"some string".doLater`**.

5. Evaluation

To evaluate the ability of Scala and Groovy to be DSL hosts, I will compare the DSLs created in them based on their features. Then I will compare Scala and Groovy based on their capabilities to allow existing language functionality to be augmented.

5.1. DSL Comparison

5.1.1. Bind

First I will compare the two DSLs based on the different types of incremental dependency-based evaluation features that I have implemented.

5.1.1.1. Bound Property

For bi-directional binding of properties the syntax in the two DSLs as well as the original JavaFX syntax is given below where **`foo`** and **`bar`** are two properties being bound to each other.

```
JavaFX syntax: foo bind bar
DSL in Scala:  foo bind bar
DSL in Groovy: foo.bind bar
```

Since clarity and expressiveness is an important requirement for a DSL, the syntax of usage becomes important. Scala syntax looks more like a DSL as compared to Groovy's because it allows methods to be used as infix operators. Both languages allow single parameter methods to be used without parentheses which makes for better DSL syntax. The two DSL syntax are pretty much the same as JavaFX syntax for bind except we have to declare the two properties to be instances of class **`BoundProp`**. The proper syntax to set the value of a property that is bound is given below.

```
JavaFX syntax: bar = "new value"
DSL in Scala:  bar:= "new value"
DSL in Groovy: bar.set "new value"
```

Neither language allowed the overloading of assignment operator (=). One trick in Groovy would have been to intercept the assignment call and include the extra functionality there, but Groovy did not allow the interception of the assignment function call. As Scala allows creating your own operators, I was still able to achieve a nice assignment syntax. But in case of Groovy, since it does not allow operators as valid identifiers, the syntax for assignment looks like a regular function call. Also as discussed in Section 4.1.1.1 I encountered problems with Groovy generics while implementing the **`BoundProp`** class.

5.1.1.2. Bound Swing Components

Lets now look at the bi-directional bind of Swing components and compare their syntax. The variables **`field1`** and **`field2`** refer to **`JTextField`** objects and they are being bound on their **`text`** property.

```
JavaFX syntax: field1.text bind field2.text
DSL in Scala:  field1.text bind field2.text
DSL in Groovy: field1.property.text.bind field2.property.text
```

The syntax for binding in Groovy is very verbose and compromises clarity. The Scala syntax is succinct and clearly shows that we are binding the text property of **`field1`** to the text property of **`field2`**.

For Scala we needed to include the **`import boundUtilities.BoundJTextField._;`** statement for usage of bind functionality on **`JTextField`** and then the declaration of text field would be the same as in JavaFX. For Groovy we did not need any additional statements and just having the DSL package in the class path automatically adds the bind functionality to **`JTextField`**. Also in Scala, to avoid the problem of too many implicit conversions available in a scope as discussed in Section 5.2, we can limit the scope of the implicit conversion in this context by importing just in the required scope as shown below.

```
def main(args: Array[String]) : Unit = {
  ...
  import boundUtilities.BoundJTextField._;
  {
    val field1 = new JTextField;
    val field2 = new JTextField;
    //implicitly converting field1 and field2 to BoundJTextField
    field1.text bind field2.text;
  }
  ...
}
```

One problem I faced with the Swing component binding for the DSLs was that I had to write code for every single property of every single Swing component that I wanted to bind. My approach to create a generic code for binding of any property using JavaBeans bound property failed due to the lack inbuilt native bound properties in Java. Another issue I faced due to lack of native bound properties in Java was discussed in Section 4.2.1.2 where **`field1.setText("new value")`** did not result in any property change events being fired. For this reason changes to the text property through **`setText`** method did not update other properties dependent on it.

5.1.1.3. Bind with Expressions

For binding a property to an expression involving other properties the syntax in JavaFX and my DSL is given below. The only difference is that the properties **`bar`** and **`foo`** need to be declared as **`BoundProp`** in case of the DSL.

```
JavaFX syntax: bar bind foo + 10
DSL in Scala:  bar bind foo + 10
```

Neither Scala nor Groovy at the time provided a built in feature to handle the unevaluated expression tree. I was looking for a functionality like LISP macros to store unevaluated expressions. But the latest version of Groovy has the new AST Transformations feature which might be helpful in creating expression trees. There is also a GUI library that is being developed in Scala called Scales / ScalaFX [22]. It promises to provide JavaFX like functionality for reducing boilerplate code. But it is still its early stages and there is no documentation available on what functionalities it will provide.

The expression handling mechanism for my DSL in Scala was created by manually implementing a **`Term`** class and its subclasses as described in Section 4.1.1.4 on bind with expressions. The advantage Scala provided in implementing the expression mechanism was due to its case class and pattern matching functionality as discussed in Section 3.1.6. Since Groovy did not provide any special mechanism to implement expressions, it was felt that implementing the functionality in Groovy would not help in its evaluation as a DSL host.

Moreover, as mentioned in the bind with expressions Section of the DSL in Scala, I had to create a **`BoundPropDouble`** class to use in **`Term`** class implementation since I could not add multiplication and addition methods to the generic **`BoundProp`** class as there was no way to ensure that all types of **`BoundProp`** would support these functionalities. It would be really useful if Scala generics provided specifying method constraints. Then we could say something like the following:

```
class BoundProp[T]
        where T implements
           T + (T,T),
           T * (T,T)
{ ... }
```

Another solution would have been to have an interface for mathematical operation such as addition, multiplication etc. and have all the numeric classes such as **`Int`**, **`Double`** etc. implement it. But in the absence of any such features, I had to create a separate **`BoundPropDouble`** class that extends from **`BoundProp[Double]`** and provides **`+`** and **`*`** methods.

In the end, it was not possible to emulate the full power of bind from within a host language. The bind feature in my DSLs, unlike JavaFX bind, cannot handle expressions that call functions or have branches in them. For these, JavaFX can do magic at the compiler level that was beyond the power of my host languages to replicate. But upcoming features like Groovy's AST Tranformations which allow hooking into the compilation process to modify the Abstract Syntax Tree, might prove helpful in this area.

5.1.2. Animate Feature

Next, let us look at the syntax for the animation feature discussed in Section 2.1.2 of JavaFX.

```
JavaFX syntax: foo = [0..10] dur 3000
DSL in Scala:  foo animate (0 to 10, 3000)
DSL in Groovy: foo.animate([1,2,3,4,5,6,7,8,9,10], 3000)
```

I named the operator **`animate`** instead of **`dur`** in my DSLs as I felt that it provided a better explanation of the operator's function. I have also changed the syntax of the animation feature from being called on the Collection (as in JavaFX) to a feature on the bound property. The original syntax of our DSL (given above) used **`bind`** to update the property (**`foo`**) based on the values returned by the **`animate`** operator.

```
Old syntax: foo bind ([0 to 10] animate 1000)
```

I felt that it was a better idea for my DSL to call the animation feature on the bound property and pass the collection and time as parameters instead of calling **`animate`** with **`bind`**.

The Groovy syntax is slightly longer since we have to specify the entire list of values. Groovy does support the **`Range`** syntax [1,2,..10] but instead of generating 1,2,3,4,5,6,7,8,9,10 it generates 1,2, [3,4,5,6,7,8,9,10] due to which **`foo`** gets assigned the value 1 and 2 correctly but the third value assigned to it is [3,4,5,6,7,8,9,10] instead of 3. Therefore, I could not use it in the animate call. The Scala **`Range`** class only supports integers and it would be nice if they provided the same functionality for other numeric classes.

The animate feature in my DSLs accepts all kinds of collections. The Groovy syntax does not require any extra imports but for Scala we need the special import statement to include the bound companion class of the Swing component we want to animate.

5.1.3. Concurrency Features

The next set of features for the DSL were the concurrency related **`do`** and **`do later`** as discussed in Section 2.1.3 of JavaFX. A comparison of syntax for **`do later`** feature is given below. The syntax for **`do`** follows the same pattern.

```
JavaFX syntax:
do later{
  //some task
}

DSL in Scala:
doLater{
  //some task
}
```

DSL in Groovy:

```
use(ThreadUtilCategory){ //or Object.doLater
  doLater{ ... }
}
```

Once again Groovy did not provide the functionality to create a simple control statement. We either have to include the **`use`** statement or call **`Object.doLater`** as discussed in Section 4.2.3 of the DSL in Groovy. Scala requires **`import boundUtilities.ExtendedObject._;`** to include the extra concurrency functionality.

5.2. Comparison of Language Features

In this part I will discuss Scala's views/implicit conversion and Groovy's meta programming facilities. Then a comparison of features that help create a more DSL like syntax follows. In the end I present a few comparisons based on my experience of working with the two languages.

It was surprising that the DSL in Groovy turned out to have more issues than the one in Scala. Surprising because when I initially compared Scala and Groovy on their features that allow language extensions, Groovy seemed much more promising with its very rich meta programming feature set. Scala only had views/implicit conversion but Groovy provided multiple ways of enhancing or modifying an existing class. Groovy had categories, and **`ExpandoMetaClass`** and **`DelegatingMetaClass`** which allowed not only to add new methods but also intercept calls to existing methods and even allowed addition of methods at run time using method missing functionality. Groovy has also had practical success with creating DSLs such as GORM [23] and builders[10]. But in the end for **my specific** DSL Scala provided better syntax than I could achieve in Groovy.

One thing worth mentioning here is that Scala's implicit conversion feature can lead to issues when scaled. It is easy to keep track of implicit conversions in a limited scope but as the scope gets bigger and we have multiple implicit conversions available it gets harder to keep track of which conversion is being applied at which point. During the creation of DSL in Scala I encountered the following issue. Consider the code

```
class Bar(o: Object){ //class declaration and constructor
  //...
}

class Foo extends Bar{
  //...
}
```

Class **`Foo`** should not be allowed to extend from **`Bar`** without passing anything to the parent **`Bar`** which only provides a single, one parameter, constructor. What happens is that Scala applies an implicit conversion to convert a parameter-less class **`Bar`** to **`Bar()`** with an empty parameter list and then converts the parameter list to a single tuple parameter **`Bar(())`** so **`Bar`** gets created with a **`Scala.Unit`** or **`()`** as a parameter. I believe

that Scala needs to provide an easy way to track implicit conversions to see which conversions are being applied to which statement in the code.

Besides the major language extension providing features there were many small features in Scala that allowed for better DSL creation than Groovy. Some of these helped provide better DSL syntax in Scala and others were general issues I encountered with Groovy that were not faced in Scala. Below is an overview of some of those features.

Table 1. Comparison of some features enabling better DSLs in Scala versus Groovy

Scala	Groovy
Operators as valid identifiers	Not allowed
Methods as infix operators	Not allowed
Parentheses less method	Allowed
Good generics support	Problems in generics
Supports inner classes	Does not allow inner classes

Lastly, Scala provides much better documentation and has many documents ranging from tutorials by example and language references to details on how a certain feature works. I found that documentation on Groovy was sparse and it was hard to understand the working of certain features. I could not find a good comparison of `ExpandoMetaClass` and `DelegatingMetaClass` and which class was better suited for use in which scenario. The Scala mailing list was also very active and quickly provided good responses which were not only from users but also from the developers of Scala. My experience with Groovy's mailing list was that most responses were from users and did not always solve the issue in question. Scala also had a very helpful IRC channel available as compared to Groovy's IRC presence. The IDE support is also better in Scala despite the fact that Scala's Eclipse plugin is an experiment on creating a plugin Scala. These issues, though not directly indicative of the worth of a language for DSL creation, certainly influence a developer. They have an effect on how much a developer can achieve in the language and how smooth the development effort will be.

6. Conclusion

In this project, I set out to evaluate and compare the dynamic features of Scala and Groovy and assess their ability to be DSL hosts. To achieve this I created JavaFX like DSLs for rich user interface application development in each of the two languages. The DSLs had powerful features like bind and others which allowed easier syntax for common operation in the DSL domain and less boiler plate code.

The results of the project were surprising compared to original expectations. Groovy with its MOP based rich feature set for DSL creation did not perform as well as Scala. With Groovy's MOP we could add new methods, replace or augment existing methods and even create new methods at runtime that were missing when the method call was made. It also allowed interception of calls to properties. For all these reasons MOP seemed very flexible for creating DSLs. One negative aspect of MOP is the lack of compile-time type checking. Scala does not support MOPs but its implicit conversion/views feature enabled us to create our DSL and provided compile-time type checking too. Scala also provided a few other features such as allowing operators to be valid identifiers, and methods to be used as infix operators etc. which enabled better DSL syntax than Groovy.

When comparing the languages on ease of use, Scala again fares better due to a good IDE and a very helpful support network. I also encountered more bugs in Groovy as compared to Scala during the development of the DSLs, though bugs at this stage are natural in developing languages.

The lack of inbuilt properties in Java was an issue I faced on more than one occasion during the DSL development. The hope is that a future version of Java will have support for native bound properties allowing a generic solution for creating bound properties for our DSLs.

Nonetheless I believe Groovy has many powerful features that would allow it to be a good DSL host. My DSL was just one particular case and not indicative of all types of features a DSL might require. Groovy has had practical success in DSLs requiring nested structures like XML, ANT and others. Groovy is still in its early stages and I think some of the issues that I faced like problems with generics etc. will be resolved soon. With the much wider interest in Groovy the documentation situation might also get better. With the IDE's of both languages becoming better everyday, developing in them should also get easier.

Interestingly, during the course of my project, another project has started from the Groovy team called the SwingBuilder which has many of the same features as my DSL but the project is much more extensive and is creating a declarative syntax for user interface creation. It will be interesting to see the results of that project.

My implementation of bind with expressions revealed that in some cases the bind feature cannot be easily dealt with in a DSL created in a host language. In those cases a new language that has access to the compiler has more power. It shows a good reason why

Sun created JavaFX as a new language as compared to implementing it inside a host language.

The project work was challenging since all three languages I used were recent and evolving with highly fluid feature sets. Many times I had to download the latest source code for the language and build it myself just so I could run some feature I was using in my DSL that was still in its testing phase. It was normal for a prototype like the interpreter based JavaFX to be changing so rapidly, but chasing a moving target came with its own set of challenges. Some of the features available in the interpreted JavaFX that I implemented in my DSL were “dumbed down” to accommodate them in the compiled JavaFX version. An example is the automatic bi-directional binding that is now unidirectional by default and requires further specification in the compiler version of JavaFX to make it bi-directional. The Integrated Development Environments (IDEs) for the languages were also in their nascent stages, posing further challenges. Since the languages were new, there were not many existing samples of DSLs in the languages. Therefore, the project involved a lot of innovate utilization of the language features.

In the end the project provides a useful comparison that shows how a language requires some simple features in addition to language extension features like meta programming, to be a good DSL host. It also highlights some issue areas in both languages that if resolved would make development in these languages much smoother. But the results are promising in that all the features of JavaFX selected were implementable in our DSL albeit a few syntax changes. This proves that instead of creating a DSL from scratch, it will be worth while to look into these languages for all DSL needs in the future.

References

[1] Henry, K. 2006. A crash overview of groovy. Crossroads, 12, 3, ACM Press, May 2006.

[2] JavaFX. Sun Microsystems. (Accessed September 2007). http://www.sun.com/software/javafx/index.jsp

[3] Kiczales, G., des Rivières, J., and Bobrow, D. G. 1991. The Art of the Metaobject Protocol. MIT Press, 1991.

[4] Mernik, M., Heering, J., and Sloane, A. M. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)* 37, 4 (Dec. 2005), 316-344.

[5] Odersky et al. 2007. A Tour of the Scala Programming Language. Programming methods laboratory EPFL, May 2007. Available from http://www.scalalang.org/docu/files/ScalaTour.pdf

[6] Odersky, M. and Zenger, M. 2005. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 41-57.

[7] Odersky, M. 2006. The Scala experiment: can we provide better language support for component systems?. In Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, South Carolina, USA, January 11 - 13, 2006). POPL '06. ACM, New York, NY, 166-167.

[8] The JavaFX Script Programming Language. (Accessed September 2007). https://openjfx.dev.java.net/JavaFX_Programming_Language.html

[9] van Deursen, A., Klint, P., and Visser, J. 2000. Domain-specific languages: an annotated bibliography. SIGPLAN Not. 35, 6 (Jun. 2000), 26-36.

[10] Groovy, An agile dynamic language for the Java Platform. (Accessed November 2007). http://groovy.codehaus.org

[11] Chris Oliver's Weblog. (Accessed April 26, 2008). http://blogs.sun.com/chrisoliver/entry/programming_animations_in_fx

[12] Horstmann, C. and Cornell, G. (2007). *Core Java: Volume I – Fundamentals 8th Edition*. Santa Clara: Prentice Hall.

[13] Horstmann, C. and Cornell, G. (2008). *Core Java: Volume II – Advanced Features 8th Edition*. Santa Clara: Prentice Hall.

[14] Odersky, M. 2006. An Overview of the Scala Programming Language, Second Edition. Programming methods laboratory EPFL, Switzerland. Available from http://www.scala-lang.org/docu/files/ScalaOverview.pdf

[15] Schinz, M., Haller, P. 2007. A Scala Tutorial for Java Programmers, Version 1.2. Programming methods laboratory EPFL, Switzerland. Available from http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

[16] Learning JavaFX Script, Part 1. (Accessed April 26, 2008). http://java.sun.com/developer/technicalArticles/scripting/javafxpart1/

[17] The JavaFX Script Programming Language Reference. (Accessed May 13, 2008). http://openjfx.java.sun.com/current-build/doc/binding.html

[18] Richter, J. (2006). *CLR VIA C# 2nd Edition*. Redmond: Microsoft Press.

[19] Quaere Home. (Accessed May 20, 2008). http://quaere.codehaus.org/

[20] Kijaro Project Home. (Accessed May 20, 2008). https://kijaro.dev.java.net/

[21] Property Specification, 3rd Draft. (Accessed May 20, 2008). http://docs.google.com/View?docid=dfhbvdfw_1f7mzf2

[22] Scales Project Home. (Accessed May 20, 2008). http://tools.assembla.com/scales

[23] Grails Object Relational Mapping (GORM). (Accessed May 20, 2008). http://grails.org/gorm