1. Introduction

This project is the expansion of the hw1. This project now allows users to define classes, create instances, invoke methods, and perform fuzzy logic operations on numbers and sets. The fuzzy gates and fuzzy sets are defined in classes.

2. Language Design

Class

The Class case class represents the blueprint of an object in the DSL. Here are the explanations for each components of a class.

case class Class(

name: String,

superClass: Option[Class] = None,

methods: List[Method] = Nil,

vars: List[ClassVar] = Nil

)

Name: A string identifies the class

SuperClass: An optional reference to a superclass. Superclass allows possible inheritance.

Methods: A list of Method instances defining the name, params used in the class, and the actual operations used in this method.

Vars: A list of ClassVar instances representing the variables used in the class.

Class Inheritance

Class inheritance is applied through the superClass parameter in the Class definition. By specifying a superclass, a class can inherit methods and variables from its parent. Here is an example of a derive class.

val baseClass = Class(

name = "Base",

methods = List(

Method(

name = "multMethod",

params = List(Parameter("p1", "double"), Parameter("p2", "double")),

body = MULT(Input("p1"), Input("p2"))

)

)

)

val derivedClass = Class(

name = "Derived",

superClass = Some(baseClass),

methods = List(

Method(

name = "xorMethod",

params = List(Parameter("p1", "double"), Parameter("p2", "double")),

body = XOR(Input("p1"), Input("p2"))

)

)

)

CreateNew

This DSL uses CreateNew to create a new instance of a class. It is used in other creation methods

case class CreateNew(clazz: Class)

CreateInstance

The instance creation method uses CreateNew. It gets the contents and the name of the clazz as instance.

def CreateInstance(clazz: Class): CreateNew = {

val instance = CreateNew(clazz)

classInstances += (clazz.name -> instance)

instance

}

Method Invocation

This function is used to invoke a method defined within a class on a specific instance. It first tries to retrieve the scope by the class name. If it doesn't exist, it returns an empty map. Then For each argName in argNames, it tries to find the corresponding value in scope. If the value is found, it creates a key-value pair, otherwise it would throw an error. Finally, the method would be evaluated if everything goes well.

def Invoke(instance: CreateNew, methodName: String, argNames: List[String]): FuzzyValue = {

val scope = instanceScope.getOrElse(instance.clazz.name, Map())

val args: Map[String, FuzzyValue] = argNames.map { argName =>

scope.get(argName) match {

case Some(value) => argName -> value

case None => throw new IllegalArgumentException(s"Input $argName is not defined in the scope for instance ${instance.clazz.name}")

}

}.toMap

FuzzyGateEvaluator.evaluateMethod(instance, methodName, args)

}

Parameter

Parameter case class defines the parameters accepted by a method.

case class Parameter(name: String, paramType: String)

Method

The Method case class represents the function used in a class. It includes:

Name: The identifier for the method.

Params: A list of Parameter instances defining the method's inputs.

Body: A FuzzyOperation that represents the method's logic.

Here is the example of the Method usage:

Method(

name = "intersectionMethod",

params = List(Parameter("setA", "set"), Parameter("setB", "set")),

body = Intersection(SetInput("setA"), SetInput("setB"))

)

ClassVar

ClassVar represents a variable or attribute of a class.

case class ClassVar(name: String, varType: VarType)

VarType

VarType is a sealed trait that defines the types of variables allowed in the DSL. It include String, Double and Set.

sealed trait VarType

case object StringType extends VarType

case object DoubleType extends VarType

case object SetType extends VarType

3. Evaluation

The evaluation of expressions is controlled by the FuzzyGateEvaluator. It provides evaluation methods for both fuzzy gate operations and fuzzy set operations.

Method Evaluation

When a method is invoked, the evaluator finds the method by the method name and then evaluates its body within the given scope.

def evaluateMethod(instance: CreateNew, methodName: String, args: Map[String, FuzzyValue]): FuzzyValue = {

val clazz = instance.clazz

val method = findMethod(clazz, methodName).getOrElse(throw new IllegalArgumentException(s"Method $methodName not found in class ${clazz.name}"))

evaluateMethodBody(method.body, args)

}

Method Body Evaluation

The evaluateMethodBody function matches the body of the method. It would distinguish the fuzzy gate operations and fuzzy set operations. For AlphaCut, it is a single set operation so I put it here.

private def evaluateMethodBody(body: FuzzyOperation, scope: Map[String, FuzzyValue]): FuzzyValue = {

body match {

case op: FuzzyGateOperation => FuzzyNumber(evaluateGateOperation(op, scope))

case op: FuzzySetOperation => FuzzySetValue(evaluateSetOperation(op, scope))

case AlphaCut(setOp, alphaOp) =>

val setA = evaluateSetOperation(setOp, scope)

val alphaValue = evaluateGateOperation(alphaOp, scope)

val elements = FuzzySetOperations.alphaCut(setA, alphaValue)

FuzzyString(elements.mkString(","))

}

}

Gate Operation Evaluation

Gate operations are evaluated recursively. It first checks if the scope is empty or not, and then handles the operands ADD, MULT and XOR.

private def evaluateGateOperation(op: FuzzyGateOperation, scope: Map[String, FuzzyValue]): Double = {

op match {

case Input(name, \_) =>

scope.get(name) match {

case Some(FuzzyNumber(value)) => value

case \_ => throw new IllegalArgumentException(s"Input $name is not defined as a number in the scope")

}

case ADD(a, b) => FuzzyOperations.add(evaluateGateOperation(a, scope), evaluateGateOperation(b, scope))

case MULT(a, b) => FuzzyOperations.mult(evaluateGateOperation(a, scope), evaluateGateOperation(b, scope))

case XOR(a, b) => FuzzyOperations.xor(evaluateGateOperation(a, scope), evaluateGateOperation(b, scope))

}

}

Set Operation Evaluation

Set operations are designed similarly.

private def evaluateSetOperation(op: FuzzySetOperation, scope: Map[String, FuzzyValue]): FuzzySet = {

op match {

case SetInput(name) =>

scope.get(name) match {

case Some(FuzzySetValue(set)) => set

case \_ => throw new IllegalArgumentException(s"Set $name is not defined as a set in the scope")

}

case Union(a, b) => FuzzySetOperations.union(evaluateSetOperation(a, scope), evaluateSetOperation(b, scope))

case Intersection(a, b) => FuzzySetOperations.intersection(evaluateSetOperation(a, scope), evaluateSetOperation(b, scope))

case Complement(a) => FuzzySetOperations.complement(evaluateSetOperation(a, scope))

case AddSets(a, b) => FuzzySetOperations.add(evaluateSetOperation(a, scope), evaluateSetOperation(b, scope))

case MultSets(a, b) => FuzzySetOperations.mult(evaluateSetOperation(a, scope), evaluateSetOperation(b, scope))

case XorSets(a, b) => FuzzySetOperations.xor(evaluateSetOperation(a, scope), evaluateSetOperation(b, scope))

}

}

TestGate

The usage of TestGate is like HW1. The comment after the expected result explains how those results are calculated. Here is an example:

*it* should "correctly evaluate MULT operation on fuzzy sets in a class method" in {  
 val baseSetClass = Class(  
 name = "BaseSet",  
 methods = List(  
 Method(  
 name = "multMethod",  
 params = List(Parameter("setA", "set"), Parameter("setB", "set")),  
 body = MultSets(SetInput("setA"), SetInput("setB"))  
 )  
 )  
 )  
  
 val instance = *CreateInstance*(baseSetClass)  
  
 *ScopeInstance*(instance, "setA", FuzzySetValue(FuzzySet("A", List(Element("x1", 0.6), Element("x2", 0.8)))))  
 *ScopeInstance*(instance, "setB", FuzzySetValue(FuzzySet("B", List(Element("x1", 0.5), Element("x3", 0.4)))))  
  
 val result = *Invoke*(instance, methodName = "multMethod", argNames = List("setA", "setB"))  
  
 result match {  
 case FuzzySetValue(set) =>  
 set.elements shouldEqual List(  
 Element("x1", 0.3), //0.6\*0.5  
 Element("x2", 0.0),  
 Element("x3", 0.0)  
 )  
 case \_ => fail("Expected a FuzzySetValue")  
 }  
}

In this example, the class is defined to call the multiply function for fuzzy sets. SetA does not have x3 and SetB does not have x2. Therefore, x3 in setA and x2 in setB are considered and 0.0. Then the multiply result for x1 is 0.6\*0.5=0.3. x2 and x3 are both 0.

4. Semantics of the Language

Class Definitions: Encapsulate related methods and variables. The super type supports class inheritance, like the derived class.

Instance Creation: The instance contains the content of classes. They could be called by method invocation by finding the name.

Method Invocation: Given provided arguments, method invocation could call the corresponding methods in classes.

Scope Management: Variables and their values are defined within scopes. Method invocation could use similar scope managements for all defined scopes.

Expression Evaluation: The evaluation body distinguishes fuzzy gate and fuzzy set operations and evaluates each operand recursively.

Type Handling: The DSL could distinguish between different variable types (DoubleType, StringType, SetType). And early error throw ensures type safety during evaluation.

5. Type system

In this project, int value and floating point gate value between 0 to 1 are all converted into double. The gate value could also be a string, but string value does not participate in the fuzzy operations.