1. Introduction

This project extends the previous fuzzy logic DSL in hw2 by introducing partial evaluation functions and adding conditional functions such as GREATER\_EQUAL, GREATER\_EQUAL\_SET, IFTRUE, and ELSERUN. Now this project could perform partial evaluation for both fuzzy gate and fuzzy set operations.

2. Language Design

Partial Evaluation Design

Partial evaluation is implemented through the partialEvaluate method defined in the FuzzyExpression trait and its subclasses.

sealed trait FuzzyExpression {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression

}

Each operation (include both gate and set operations) can partially evaluate expressions with the mapping variable names to their values.

Partial Evaluation of Expressions

FuzzyNumber: Returns itself since it could not be evaluated further.

Variable: Looks up its value in the list or returns itself.

Input: Get the corresponding value from the scope or remains unchanged.

Operations: Partially evaluate the operands and attempt to compute the result if possible.

Here is the example implementation of ADD:

case class ADD(a: FuzzyExpression, b: FuzzyExpression) extends FuzzyGateOperation {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

val left = a.partialEvaluate(env)

val right = b.partialEvaluate(env)

(left, right) match {

case (FuzzyNumber(v1), FuzzyNumber(v2)) => FuzzyNumber(math.min(v1 + v2, 1.0))

case \_ => ADD(left, right)

}

}

}

The match function enables the add to compute the result if it is possible to perform ADD operand.

Similar logic is used for set operations:

case class Union(a: FuzzySetExpression, b: FuzzySetExpression) extends FuzzySetOperation {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

val left = a.partialEvaluate(env).asInstanceOf[FuzzySetExpression]

val right = b.partialEvaluate(env).asInstanceOf[FuzzySetExpression]

(left, right) match {

case (FuzzySetValue(set1), FuzzySetValue(set2)) =>

FuzzySetValue(FuzzySetOperations.union(set1, set2))

case \_ =>

Union(left, right)

}

}

}

SetInput expressions are partially evaluated by looking up the set.

Conditional Constructs

GREATER\_EQUAL and GREATER\_EQUAL\_SET

The GREATER\_EQUAL and GREATER\_EQUAL\_SET functions allow for fuzzy comparisons between numbers and sets.

GREATER\_EQUAL: Compares two fuzzy numbers.

case class GREATER\_EQUAL(a: FuzzyExpression, b: FuzzyExpression) extends FuzzyOperation {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

val left = a.partialEvaluate(env)

val right = b.partialEvaluate(env)

(left, right) match {

case (FuzzyNumber(v1), FuzzyNumber(v2)) =>

FuzzyBoolean(v1 >= v2)

case \_ =>

GREATER\_EQUAL(left, right)

}

}

}

GREATER\_EQUAL\_SET: Compares the sum of membership values of two fuzzy sets.

case class GREATER\_EQUAL\_SET(a: FuzzySetExpression, b: FuzzySetExpression) extends FuzzyOperation {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

val left = a.partialEvaluate(env)

val right = b.partialEvaluate(env)

(left, right) match {

case (FuzzySetValue(set1), FuzzySetValue(set2)) =>

val sum1 = set1.elements.map(\_.value).sum

val sum2 = set2.elements.map(\_.value).sum

FuzzyBoolean(sum1 >= sum2)

case \_ =>

GREATER\_EQUAL\_SET(left.asInstanceOf[FuzzySetExpression], right.asInstanceOf[FuzzySetExpression])

}

}

}

IFTRUE, THENEXECUTE, and ELSERUN

These functions enable conditional execution based on the result of a condition.

IFTRUE: Represents a conditional expression that evaluates one of two cases based on the result of the FuzzyBoolean.

case class IFTRUE(condition: FuzzyExpression, thenExpr: THENEXECUTE, elseExpr: ELSERUN) extends FuzzyExpression {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

val condResult = condition.partialEvaluate(env)

condResult match {

case FuzzyBoolean(true) =>

thenExpr.partialEvaluate(env)

case FuzzyBoolean(false) =>

elseExpr.partialEvaluate(env)

case \_ =>

IFTRUE(condResult, thenExpr, elseExpr)

}

}

}

THENEXECUTE: Let the expression to execute if the condition is true.

case class THENEXECUTE(expr: FuzzyExpression) extends FuzzyExpression {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

expr.partialEvaluate(env)

}

}

ELSERUN: Let the expression to execute if the condition is false.

case class ELSERUN(expr: FuzzyExpression) extends FuzzyExpression {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

expr.partialEvaluate(env)

}

}

FuzzyBoolean

Represents a boolean value in the DSL.

case class FuzzyBoolean(value: Boolean) extends FuzzyValue {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = this

}

Method Invocation

Method invocation is updated to return an InvokeMethod expression, which can be partially evaluated.

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

val args: List[(String, FuzzyExpression)] = argNames.map { argName =>

(argName, Input(argName))

}

InvokeMethod(instance, methodName, args)

}

case class InvokeMethod(instance: FuzzyExpression, methodName: String, args: List[(String, FuzzyExpression)]) extends FuzzyExpression {

def partialEvaluate(env: Map[String, FuzzyExpression]): FuzzyExpression = {

val evaluatedInstance = instance.partialEvaluate(env)

val evaluatedArgs = args.map { case (name, expr) => (name, expr.partialEvaluate(env)) }.toMap

evaluatedInstance match {

case CreateNew(clazz, instanceName) =>

val methodOption = findMethod(clazz, methodName)

methodOption match {

case Some(method) =>

val methodEnv = evaluatedArgs ++ env

val instanceVars = clazz.vars.map { classVar =>

val varName = s"${instanceName}.${classVar.name}"

(classVar.name, env.getOrElse(varName, Variable(varName)))

}.toMap

method.body.partialEvaluate(methodEnv ++ instanceVars)

case None => throw new IllegalArgumentException(s"Method $methodName not found in class ${clazz.name}")

}

case \_ => throw new IllegalArgumentException(s"Cannot invoke method on non-class instance: $evaluatedInstance")

}

}

}

3. Evaluation

evaluateExpression

The evaluateExpression function is fully updated in this project to handle partial evaluation for fuzzy gate evaluations.

def evaluateExpression(expr: FuzzyExpression, env: Map[String, FuzzyExpression]): (FuzzyExpression, Map[String, FuzzyExpression]) = expr match {

case FuzzyNumber(value) => (FuzzyNumber(value), env)

case Variable(name) =>

val parts = name.split("\\.")

if (parts.length == 2) {

val instanceName = parts(0)

val variableName = parts(1)

val fullVarName = s"$instanceName.$variableName"

val value = env.getOrElse(fullVarName, Variable(fullVarName))

(value, env)

} else {

val value = env.getOrElse(name, Variable(name))

(value, env)

}

case ADD(a, b) =>

val (left, env1) = evaluateExpression(a, env)

val (right, env2) = evaluateExpression(b, env1)

(left, right) match {

case (FuzzyNumber(v1), FuzzyNumber(v2)) =>

(FuzzyNumber(FuzzyOperations.add(v1, v2)), env2)

case \_ =>

(ADD(left, right), env2)

}

case MULT(a, b) =>

val (left, env1) = evaluateExpression(a, env)

val (right, env2) = evaluateExpression(b, env1)

(left, right) match {

case (FuzzyNumber(v1), FuzzyNumber(v2)) =>

(FuzzyNumber(FuzzyOperations.mult(v1, v2)), env2)

case \_ =>

(MULT(left, right), env2)

}

case XOR(a, b) =>

val (left, env1) = evaluateExpression(a, env)

val (right, env2) = evaluateExpression(b, env1)

(left, right) match {

case (FuzzyNumber(v1), FuzzyNumber(v2)) =>

(FuzzyNumber(FuzzyOperations.xor(v1, v2)), env2)

case \_ =>

(XOR(left, right), env2)

}

case GREATER\_EQUAL(a, b) =>

val (left, env1) = evaluateExpression(a, env)

val (right, env2) = evaluateExpression(b, env1)

(left, right) match {

case (FuzzyNumber(v1), FuzzyNumber(v2)) => (FuzzyBoolean(v1 >= v2), env2)

case \_ => (GREATER\_EQUAL(left, right), env2)

}

case GREATER\_EQUAL\_SET(a, b) =>

val (left, env1) = evaluateSetExpression(a, env)

val (right, env2) = evaluateSetExpression(b, env1)

(left, right) match {

case (FuzzySetValue(setA), FuzzySetValue(setB)) =>

val cardinalityA = setA.elements.map(\_.value).sum

val cardinalityB = setB.elements.map(\_.value).sum

(FuzzyBoolean(cardinalityA >= cardinalityB), env2)

case \_ =>

(GREATER\_EQUAL\_SET(left.asInstanceOf[FuzzySetExpression], right.asInstanceOf[FuzzySetExpression]), env2)

}

case IFTRUE(condition, thenExec, elseRun) =>

val (condResult, env1) = evaluateExpression(condition, env)

condResult match {

case FuzzyBoolean(true) =>

evaluateExpression(thenExec.expr, env1)

case FuzzyBoolean(false) =>

evaluateExpression(elseRun.expr, env1)

case \_ =>

(IFTRUE(condResult, thenExec, elseRun), env1)

}

case Assign(variable, expr) =>

val (evaluatedExpr, env1) = evaluateExpression(expr, env)

val updatedEnv = env1 + (variable.name -> evaluatedExpr)

(evaluatedExpr, updatedEnv)

case THENEXECUTE(expr) =>

evaluateExpression(expr, env)

case ELSERUN(expr) =>

evaluateExpression(expr, env)

case InvokeMethod(instanceExpr, methodName, args) =>

val (evaluatedInstance, env1) = evaluateExpression(instanceExpr, env)

evaluatedInstance match {

case instance: CreateNew =>

val instanceName = instance.instanceName

// Evaluate arguments

val (evaluatedArgs, env2) = args.foldLeft((List.empty[(String, FuzzyExpression)], env1)) {

case ((argList, envAcc), (argName, argExpr)) =>

val (evaluatedArg, envNext) = evaluateExpression(argExpr, envAcc)

(argList :+ (argName, evaluatedArg), envNext)

}

val method = findMethod(instance.clazz, methodName).getOrElse(

throw new IllegalArgumentException(s"Method $methodName not found in class ${instance.clazz.name}")

)

val methodEnv = env2 ++ evaluatedArgs.toMap

val instanceVars = instance.clazz.vars.map { classVar =>

val varName = s"${instanceName}.${classVar.name}"

(classVar.name, env2.getOrElse(varName, Variable(varName)))

}.toMap

val methodEnvWithInstance = methodEnv ++ instanceVars

val (result, methodEnvUpdated) = evaluateExpression(method.body, methodEnvWithInstance)

val updatedInstanceVars = methodEnvUpdated.collect {

case (varName, value) if instanceVars.contains(varName) =>

val instanceVarName = s"${instanceName}.${varName}"

(instanceVarName, value)

}

val updatedEnv = env2 ++ updatedInstanceVars

(result, updatedEnv)

case \_ =>

throw new IllegalArgumentException(s"Cannot invoke method on non-class instance: $evaluatedInstance")

}

case setExpr: FuzzySetExpression =>

evaluateSetExpression(setExpr, env)

case Input(name) =>

env.get(name) match {

case Some(value) => (value, env)

case None => (Input(name), env)

}

case \_ => (expr, env)

}

Gate and set operations are implemented similarly in evaluateGateOperation and evaluateSetExpression. The code is too long so I don’t include them here.

This function could resolve inputs and variables. As for Conditional Expressions, they are evaluated by first evaluating the condition and then jump to the correct branch.

I had put the implementation of partial evaluation in this part to prevent some errors, as well as some rounding problems. but partial evaluation has already been implemented in the operations. There should be some way to cut the redundancy but I could not find it.

Partial Evaluation

Partial evaluation allows expressions to be evaluated as much as possible given the available information. Undefined variables remain as Input expressions.

val expr = ADD(FuzzyNumber(0.3), Input("x"))

val partiallyEvaluatedExpr = expr.partialEvaluate(Map.empty)

// Result: ADD(FuzzyNumber(0.3), Input("x"))

X is not defined so this expression is partially evaluated.

Some other examples would be mentioned is test part.

4. Test Design

The first part of the test is to test if all operations could work normally without introducing evaluation. The next part is to test evaluation and condition.

Testing Class Methods with Partial and Full Evaluation

it should "partially and fully evaluate a class method" in {

val baseClass = Class(

name = "Base",

methods = List(

Method(

name = "computeMethod",

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

body = MULT(FuzzyNumber(0.5), ADD(Input("p1"), Input("p2")))

)

)

)

val instance = CreateInstance(baseClass, "instance1")

val methodExpr = Invoke(instance, methodName = "computeMethod", argNames = List("p1", "p2"))

// Partial evaluation with only p1 defined

val partiallyEvaluatedMethod = methodExpr.partialEvaluate(Map("p1" -> FuzzyNumber(0.4)))

partiallyEvaluatedMethod shouldEqual MULT(FuzzyNumber(0.5), ADD(FuzzyNumber(0.4), Input("p2")))

// Full evaluation with both p1 and p2 defined

val (result, \_) = FuzzyGateEvaluator.evaluateExpression(methodExpr, Map("p1" -> FuzzyNumber(0.4), "p2" -> FuzzyNumber(0.3)))

result shouldEqual FuzzyNumber(0.35)

}

For Partial Evaluation, one value is FuzzyNumber 0.4, the other is Input, cannot compute add function. So the result for addition is ADD(FuzzyNumber(0.4), Input("p2")). And it could not compute multiplication since one value is not a FuzzyNumber.

The final result: MULT(FuzzyNumber(0.5), ADD(FuzzyNumber(0.4), Input("p2")))

For full evaluation, by Map("p1" -> FuzzyNumber(0.4), "p2" -> FuzzyNumber(0.3)), the expression becomes MULT(FuzzyNumber(0.5), ADD(FuzzyNumber(0.4), FuzzyNumber(0.3))). Then the result is 0.5\*(0.4+0.3)=0.35

Testing Conditional Expressions with Sets

it should "correctly evaluate GREATER\_EQUAL\_SET, IFTRUE, THENEXECUTE, and ELSERUN with set operations" in {

val setA = FuzzySet("SetA", List(Element("x", 0.5), Element("y", 0.7)))

val setB = FuzzySet("SetB", List(Element("x", 0.6), Element("y", 0.4)))

val setC = FuzzySet("SetC", List(Element("x", 0.8), Element("y", 0.2)))

val setD = FuzzySet("SetD", List(Element("x", 0.3), Element("y", 0.9)))

val env = Map(

"A" -> FuzzySetValue(setA),

"B" -> FuzzySetValue(setB),

"C" -> FuzzySetValue(setC),

"D" -> FuzzySetValue(setD)

)

val expr = IFTRUE(

GREATER\_EQUAL\_SET(SetInput("A"), SetInput("B")),

THENEXECUTE(

Assign(Variable("resultSet"), Union(SetInput("A"), SetInput("C")))

),

ELSERUN(

Assign(Variable("resultSet"), Intersection(SetInput("B"), SetInput("D")))

)

)

// Evaluate the expression

val (result, finalEnv) = FuzzyGateEvaluator.evaluateExpression(expr, env)

// Retrieve the resultSet from the environment

finalEnv.get("resultSet") match {

case Some(FuzzySetValue(resultSet)) =>

// Check if the condition was true or false based on the sums

val sumA = setA.elements.map(\_.value).sum // 0.5 + 0.7 = 1.2

val sumB = setB.elements.map(\_.value).sum // 0.6 + 0.4 = 1.0

if (sumA >= sumB) {

// Condition is true; resultSet should be Union of A and C

resultSet.elements should contain allOf (

Element("x", 0.8), // max(0.5, 0.8)

Element("y", 0.7) // max(0.7, 0.2)

)

} else {

// Condition is false; resultSet should be Intersection of B and D

resultSet.elements should contain allOf (

Element("x", 0.3), // min(0.6, 0.3)

Element("y", 0.4) // min(0.4, 0.9)

)

}

case \_ =>

fail("resultSet not found or not a FuzzySetValue")

}

}

Evaluate Condition GREATER\_EQUAL\_SET(SetInput("A"), SetInput("B")):

sumA = 0.5 + 0.7 = 1.2

sumB = 0.6 + 0.4 = 1.0

evaluate THENEXECUTE

Evaluate Expression: Assign(Variable("resultSet"), Union(SetInput("A"), SetInput("C")))

Perform Union operation:

For element "x":

Values: 0.5 (from setA), 0.8 (from setC)

max(0.5, 0.8) = 0.8

For element "y":

Values: 0.7 (from setA), 0.2 (from setC)

max(0.7, 0.2) = 0.7

The result set is FuzzySet("SetA\_UNION\_SetC", List(Element("x", 0.8), Element("y", 0.7)))

5. Type system

The type supported in this project is the same as hw2. 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.