The implementation of partial evaluation in the Frizzy language focuses on handling expressions with unknown variables, ensuring that the evaluator correctly processes such cases while preserving the language's semantics.

Semantics of Partial Evaluation and Reduction Rules:
I decided that if an expression contains unknown variables, the evaluator should partially evaluate it, leaving the unknown parts unevaluated. For arithmetic operations like addition and multiplication, if both operands are known values, the operation is computed. If not, the operation is reconstructed with the evaluated operands.
scala
Copy code

```
case Add(lhs, rhs) =>
  val leftEval = eval(lhs)
  val rightEval = eval(rhs)
  (leftEval, rightEval) match
    case (l: Int, r: Int) => l + r
    case _ => Add(asExpOperation(leftEval), asExpOperation(rightEval))
```

1. This approach ensures that known computations are performed, and expressions with unknowns remain as is for future evaluation.

Designing the Language with Conditional Constructs:
I included the `IFTRUE` construct to handle conditionals. When the condition can be fully evaluated to `true` or `false`, the corresponding branch is executed. If the condition cannot be fully evaluated due to unknown variables, the entire `IFTRUE` expression remains unevaluated.
scala
Copy code

```
case IFTRUE(condition, thenBranch, elseBranch) =>
  val condEval = eval(condition)
  condEval match
    case true => eval(thenBranch)
    case false => eval(elseBranch)
    case _ => IFTRUE(asExpOperation(condEval), thenBranch, elseBranch)
```

2.

Partial Evaluation of Method Invocations on Class Instances:
I added logic to handle partial evaluation of methods invoked on class instances. When a method is called with known arguments, it's evaluated normally. If arguments are unknown, the method body is partially evaluated, and any assignments are stored as expressions in the instance's environment.

scala

```
case InvokeMethod(instanceName, methodName, args) =>
  val instanceInfo = instanceRegistry.getOrElse(instanceName, throw
new Exception(s"Instance '$instanceName' not found"))
  val methodInfo = lookupMethod(instanceInfo.className, methodName)
  val methodEnv = mutable.Map[String, Any]()
  methodInfo.params.zip(args).foreach { case (param, argExp) =>
    methodEnv.update(param.name, eval(argExp))
  }
  val envStack = summon[EnvironmentStack]
  envStack.push((s"method_${methodName}", methodEnv))
  envStack.push((s"instance_${instanceName}", instanceInfo.variables))
  val result = eval(methodInfo.body)
  envStack.pop()
  envStack.pop()
  result
```

3.

Implementing Reduction Rules in the Evaluation Function:
THe `eval` function incorporates reduction rules that attempt to simplify expressions whenever possible. For example, in multiplication, if both operands are integers, the multiplication is performed. If not, the expression is reconstructed.
scala

```
case Multiply(lhs, rhs) =>
  val leftEval = eval(lhs)
  val rightEval = eval(rhs)
  (leftEval, rightEval) match
    case (l: Int, r: Int) => l * r
    case _ => Multiply(asExpOperation(leftEval),
asExpOperation(rightEval))
```

4. This ensures that expressions are reduced as much as possible while keeping unknown parts intact.

Partial Evaluation with Dynamic Dispatch Mechanism:
I handle dynamic dispatch in method calls within inheritance hierarchies. The `lookupMethod` function traverses the class hierarchy to find the appropriate method to invoke. During partial evaluation, if the method contains unknown variables, it is partially evaluated accordingly.
scala

```
def lookupMethod(className: String, methodName: String): MethodInfo =
  val classInfo = classRegistry.getOrElse(className, throw new
Exception(s"Class '$className' not found"))
  classInfo.methods.get(methodName) match
    case Some(methodInfo) => methodInfo
    case None =>
      classInfo.parent match
        case Some(parentClassName) => lookupMethod(parentClassName,
methodName)
        case None => throw new Exception(s"Method '$methodName' not
found in class hierarchy of '$className'")
```

5. This allows the evaluator to correctly handle method overriding and ensures that the most specific method is used during evaluation.