Fall 2015

Pattern-driven Programming in Scala

Huaxin Pang
San Jose State University

Pattern-driven Programming in Scala

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Huaxin Pang

December 2015

The Designated Project Committee Approves the Project Titled

Pattern-driven Programming in Scala

by

Huaxin Pang

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Jon Pearce	Department of Computer Science
Thomas Austin	Department of Computer Science
Thomas Howell	Department of Computer Science

ABSTRACT

Pattern-driven Programming in Scala

by Huaxin Pang

This is an experimental exploration of the pattern-driven programming paradigm—the sole use of pattern matching to determine the next instruction or execute. We define a pure pattern-driven programming language named PA-Scala by defining a subset of the Scala programming language, which restricts sequence control to the powerful pattern matching facilities in Scala. We use PA-Scala to explore the strengths and limitations of pattern-driven programming. By implementing a phrase structure grammar solver in PA-Scala, we show that pattern-driven programming can be used to solve general computation problems. We then implement a Prolog interpreter in PA-Scala, which demonstrates how resolution and unification can be implemented in PA-Scala. Finally we analyzed the possibility of parallel execution for PA-Scala, and show that pattern-driven programming also has the potential to achieve performance improvements by running pattern matching operations in parallel.

ACKNOWLEDGMENTS

First I would like to sincerely thank my advisor, Dr. Jon Pearce, for his continuous support and technical guidance of my master's study. Dr. Jon Pearce gave me so many detailed suggestions and instructions when I needed them, which lead to the key ideas and main structure of this thesis. Also I'm grateful for his support of my ideas and his trust in my ability.

I want to thank Dr. Thomas Austin, for his great lectures on advanced programming languages and advices on the topic, which provided great helps on my thesis.

I want to thank Dr. Thomas Howell for his strong support and great advices regarding to my study and research.

Last but not the least, I want to say great thanks to my dear wife. My wife supported me to pursue my dream and gave me strengths for conquer any difficulty.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In Dr. Jon Pearce's lecture notes [1] he adapts a classification scheme for computer architectures given by Treleaven [2] to a classification scheme for high-level language paradigms, by viewing language processors as virtual computers. Based on the control mechanisms and data mechanisms, eight types of architecture/paradigm are given:

Table 1: Programming architecture/paradigm

	Control Mechanisms			
Data Mechanisms	Control Driven	Data Driven	Demand Driven	Pattern Driven
Shared Memory	COSH	DASH	DESH	PASH
Message Passing	COME	DAME	DEME	PAME

Treleaven defined data mechanism as the way a particular argument is used by several instructions [2]. Adapting to high-level programming paradigm, data mechanism reflects the way the data is passed and shared between instructions where the concept of instruction is much broader, and includes functions, actions, events, phases, etc. There are two typical data mechanisms:

- Shared memory

  This is derived from Treleaven's low-level by-reference data mechanism [2]. In a by-reference data mechanism, an argument is shared by letting each accessing instruction have a reference to it. In shared memory architecture, data is in a centralized repository (memory, storage, database, etc) and is shared by all

the instructions [1]. This is a very efficient way to share data, but it can cause synchronization problems such as race conditions.

- Message passing

  This is related to the by-value data mechanism in [2]. In a by-value data mechanism, the argument is replicated and a separate copy is given to each accessing instruction. Message passing in high-level language programming paradigm reflects the architecture in which instructions share data by sending copies to each other. For example, in the Scala actor system actors invoke each other by sending messages. The messages could be synchronous or asynchronous.

Control mechanisms define how the execution of the operation is triggered. The following are the four control mechanisms in high-level programming paradigm:

- Control-driven

  In a control flow architecture, the currently executing instruction will determine the next instruction to execute. If it does not, the default next operation will be triggered. Control-driven can be found in familiar control structures such as IF/ELSE, WHILE and GOTO.

- Data-driven

  Data-driven is derived from the data flow computation architecture in Treleaven's classification scheme. In a data flow architecture, an instruction is enabled for execution when and only when all the input operands are available [3], which is that all the unknown operands have been replaced by results from other instructions. Directed graph are usually used to describe the flow of data between instructions for a Data-flow program [2]. As shown in Figure 1, when X is assigned a value C, the operands of "+" instruction are all ready, then

the "+" instruction will be executed. At the same time, the operands of "-" instruction are also ready, so the "-" instruction will be executed too. After these two instructions are executed, the "*" instruction will be triggered, because its operands are ready. It could be easily seen that the Data flow is inherently parallel [6].

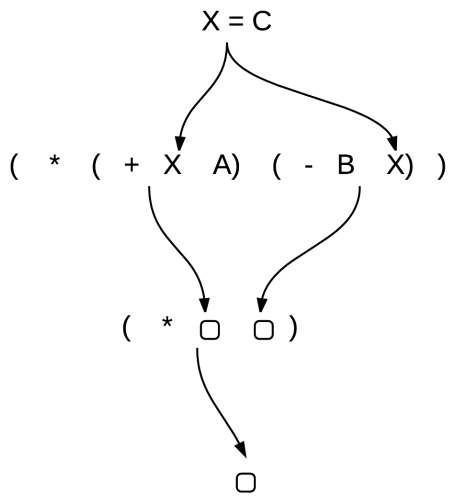

Figure 1: Data flow computation

In Data-driven paradigm, an operation is triggered when and only when the data that it relies on is ready. For another example, the CompletableFuture (Promise) object in Java 8 can define functions to execute when it completes by using *thenApply* or *thenRun* methods. So defined functions will be executed when and only when the CompletableFuture object is ready.

- Demand-driven

  Demand-driven paradigm is derived from the reduction computation architecture. Reduction programs are built with nested expressions, where the outermost expressions are evaluated first. For any instruction, its operand is evalu-

ated only when it is needed. It is also inherently parallel.

Demand-driven as a high level paradigm is an abstraction of reduction, in which operations are executed only if the results are needed. One example is lazy evaluation in Haskell [16]. When expressions are bounded to variables in Haskell, they are not evaluated immediately. Their evaluation will be deferred util the values of the variables are needed. Arguments are not evaluated when they are passed to a function, but only when their values are actually needed. The transformation operations of RDD in Spark is also an example of the demand-driven paradigm. Spark is a general-purpose clustering computing system which is widely used in big data processing. Resilient distributed dataset (RDD) is a fault-tolerant collection of elements in Spark that can be operated on in parallel. There are two types of operations that RDD supports: transformations and actions. Transformation will create a new dataset from an existing one, and actions will return a value to the driver program after running a computation. The transformation operations for RDD will just remember the transformations to some dataset, and will not get their result right away. They will only be executed when their results are needed by some action operations to return to the driver program.

- Pattern-driven

  With the development of ideas of object oriented programming, logic programming, functional language programming and some new ideas from pattern matching, pattern-driven programming emerges as another programming paradigm. Pearce added pattern-driven as a fourth mechanism to Treleaven's original scheme. The main idea of pattern-driven control is that an instruction is executed only when a given pattern matches. For example, the resolution op-

eration in Prolog's evaluation process is based on the pattern the current goal matches to determine the next rule to apply. Polymorphism in object-oriented programming languages such as C++ and Java can also be seen as examples of pattern-driven control. The type of the data can be seen as a pattern, so polymorphism will determine the concrete method to call based on the type pattern that the data matches.

However at the machine level, there are relatively few works done with the corresponding pattern-driven computation structure. The best know pattern-driven architecture was the Japanese fifth generation computer project [4][5], which was based on creating machines that directly execute Prolog. But no commercial achievement was made, and this was subsequently seen as a failure.

The Scala programming language presents a combination of a hybrid object-oriented programming language and functional programming language, with powerful facilities for many kinds of pattern matching tasks. We think that Scala could provide the key features for pattern-driven tasks, so we want to analyze the pattern matching facilities of Scala and make use of them to explore the pattern-driven programming.

In chapter 2 we analyze Scala's pattern matching capabilities and summarize the patterns that Scala can match and scenarios that Scala's pattern matching feature can be used in. We also further explored how Scala's pattern matching works, especially with extractor which could be used to define high level pattern matching.

By fully utilizing the pattern-matching facilities in Scala, in chapter 3 we define a pattern-driven programming language named as PA-Scala. PA-Scala is just a subset of Scala, but we put rigid constraints on the control commands that only pattern matching is allowed for control the execution of the program, to enforce the pure

pattern-driven programming paradigm.

In chapter 4 we implement a phrase structure grammar solver using PA-Scala. Because phrase structure grammars are Turing-equivalent, we show that PA-Scala is computationally complete for solving general computation problems. Another reason for implementing a phrase structure grammar solver is that there are many pattern matching related tasks when solving phrase structure grammar problems, which show the strength of pattern-driven programming for this kind of problems.

By implementing an interpreter for Prolog, the native language of the Japanese fifth generation computer project, we put our pattern-driven programming language, PA-Scala, to an ultimate test to explore the benefits we can get for dealing with problems that heavily rely on pattern matching. In chapter 5 we implement the Prolog interpreter step by step, going from Proplog, a small subset of Prolog, to Datalog, which adds variable feature to Proplog. At last we add a data structure feature to Datalog to finish the complete Prolog interpreter.

As pattern-driven computation is inherently parallel, in chapter 6 we discussed the potential for parallel computation in PA-Scala, a language that is not designed for parallel computation because of the root from Scala and Scala compiler it relies on. We showed that pattern-driven programming gives the program the potential to utilize multi-core systems to speed up execution.

Chapter 7 concludes the thesis by summarizing the results obtained from our exploration of pattern-driven programming. We discuss the benefits and also the limits of pattern-driven programming.

CHAPTER 2

Pattern matching in Scala

Scala is a programming language for general software applications. It was designed by Martin Odersky. Scala code is compiled to Java bytecode and run in a Java virtual machine. It is object-oriented like Java, but at same time Scala has many features of a functional programming language, which includes pattern matching. Scala has powerful built-in facilities for pattern matching, such as constant, constructor, type matching and so on. In this section, we will explore the pattern matching features in Scala and understand how they are achieved, so we can know the facilities that could be used for pattern-driven programming.

2.1 Match Expression

Scala has a *match* expression which is used to select from a number of alternatives, similar to *switch* statements in other languages such as Java, as shown in code below:

```
item match {
 case "hi" => println("hello")
 case "bye" => println("goodbye")
 case _ => println("")
}
```

The expression before *match* is called *selector*. In the body of a match expression, the expression following the *case* keyword is a pattern, and the expression following the => symbol is the action corresponding to the pattern before the => symbol.

At first glance, *match* in Scala differs with *switch* in Java in three little details:

1. Selector is written before *match*, so it's written as:

```
selector match {...}
```

compared to:

```
switch(selector) {...}
```

2. => is used to separate the condition and the corresponding code block.

3. An underscore (_) is used to specify the default case, which is a widely used wildcard symbol in Scala.

But there are also other significant differences between *match* and *switch*:

1. *match* is an expression, which means it has a return value, for example:

```
def printSound(animal:String){
    val sound = animal match {
        case "Dog" => "Woof"
        case "Cat" => "Meow"
        }
    println(sound)
}
```

2. *break* is implicit for very case clause, so it is not need to add *break* for every case clause, which means there's is no fall through from one alternative to the next. The fall through in switch expression often leads to mysterious errors that are hard to find.

3. A *MatchError* will be thrown if no case clause matches the selector. So for the *printSound* method above, if you pass in a parameter other than "Dog" or "Cat", it will throw a *MatchError*. So when using *match*, keep in mind that you should always have a default case, or you must make sure that all possible alternatives are covered.

switch statements in Java can only use ints, enum values and strings as selectors, and the alternatives for the case clause must be constants. What makes Scala's *match* expression so powerful is that *match* can applied to any type of object, and *case* can be followed by any kind of pattern. This makes the match expression in Scala not only a simple branch selection control clause but also a powerful pattern matching tool.

So let's take a look at what kinds of pattern matching that can be done using *match* expressions:

1. Constant pattern

   Constant pattern is like the case clause in *switch*, which only matches when the selector equals to the constant value. But unlike case clause in *switch*, there are many kinds of values that can be used as a constant. Any literal can be used as a constant pattern, such as 1.5, true,"Dog". Any *val* value can be a constant too. And singleton object can also be used as a constant, like *Nil*, which is a singleton object that stands for a empty list.

2. Variable pattern

   A variable name can match any object, and Scala will bind that object to the variable name, so it can b used later to refer to the matched object.

```
def printSound(animal:String){
    val sound = animal match {
        case "Dog" => "Woof"
        case "Cat" => "Meow"
        case x => "I don't know about "+x }
    println(sound)
}
```

   So when the printSound function is passed "Cow", "case x" will match it, since

x is a variable and it can match anything, and then x refers to "Cow" in the corresponding code block.

3. Wildcard pattern

   _ is the wildcard pattern which matches any object like the variable pattern. But there is no binding after the match, so it is used to match the object with a value we don't care about. It is often used as a placeholder. We have seen the usage of it as the default case.

4. Typed pattern

   A typed pattern will only match an object that is an instance of the specified type. It could be conveniently used as type test and type cast. For example:

```
abstract class Animal
class Dog extends Animal{
    def bark = println("Woof")
}
class Cat extends Animal{
    def mew = println("Meow")
}
def makeSound(animal:Animal){
    animal match{
        case d:Dog => d.bark
        case c:Cat => c.mew
        case _ => println("Huh?")
    }
}
```

   Notice how easily type tests and type casts are done in this way. If the type doesn't match, it will automatically check the next pattern, and if it matches, the type cast is done automatically so you can use it directly as a casted type after =>.

5. Tuple pattern

   Tuples can also be used as a matching pattern. For example:

```
aTuple match {
    case (first,second)  => println("2-tuple :"+first+second)
    case ("Foo",_,third) => println("3-tuple :"+"Thrid element is "+ third)
    case _               =>
}
```

Notice that one can apply patterns to the elements of a tuple. It will first check whether the object is an instance of the corresponding tuple, and then it will match against the elements with the patterns defined correspondingly inside of the parentheses.

6. Sequence pattern

   Sequences like arrays and lists can also be matched, like the tuple. For example, here is how to match against a list with 3 elements, where the first is 1, and we want to know the third element:

```
myList match{
    case List(1,_,x) => println(x)
    case _           => println("doesn't match")
}
```

   _* is used to match zero to many elements, so the following example

```
myList match{
    case List(1,_,x,_*) => println(x)
    case _              =>
}
```

   is able to get the third element of a list, if a list has three or more elements, and the first element is 1.

7. Constructor pattern

   The constructor pattern looks like a constructor, such as Dog(name,age,color). But rather than constructing, it is more like destructing when it is matched against. It consists of a name and a list of patterns in the parentheses, and these

patterns also need to be matched, which usually matched against the fields of an object. It is just a general form of tuple pattern and list pattern. So the pattern could look like Dog(_, 5, color). And patterns in the parentheses could be any pattern, including the constructer pattern itself, which means nested constructer pattern matching is possible, like Dog(_,5,Color(r,g,b)). Constructor pattern is where pattern matching in Scala becomes really powerful. We'll take a further look of constructor pattern with the case class and extractor later.

8. Pattern guard

   You can make a match expressions more precise by using a pattern guard, which is actually adding an if expression after the case clause. For example:

```
myList match{
    case List(1,_,x,_*) if x > 100=> println(x)
    case _ =>println("doesn't match")
}
```

We can see that match expression in Scala is more about pattern matching than simple branch selection in switch expression. The *match* expression is a very important tool to do pattern matching programming in Scala. Next, we will see more details about how pattern matching is implemented by understanding case class and extractor.

2.2 Case Class And Extractor

Scala allows pattern matching on objects using case classes. Case classes are just classes with case modifier in the declarations. For example:

```
case class Person(name: String,age: Int)
```

Case classes are just normal classes, but the Scala compiler adds some syntactic conveniences to them:

1. A factory method with the name of the class. So instead of new Person("Tom",12), you can use Person("Tom",12) to construct a new Person object.

2. Implicit *val* prefix for all arguments in the parameter list. So every argument will become a field of the instance.

3. Natural implementations of method *toString*, *hashCode* and *equals* are added by the Scala compiler.

4. A copy method is added to the class, which is useful for making new instance with only several attributes different.

Other than those convenience methods, case classes can be used in pattern matching, which is the greatest power of case class.

```
case class Person(name: String,age: Int)
val tom = Person("Tom",12)
tom match {
    case Person(name,age) => println(name+" is "+age)
    case _                =>
}
```

When you run this code, it will print "Tom is 12". Not only will it match against the class of the instance, but also it will extract the information out of the instance. So the class identification, class casting and property access are done together.

When doing pattern matching, the parameters in the case class can also be a pattern, which makes nested pattern matching possible.

```
case class Son(father:Person,mother:Person)
var s = Son(Person("Jack",32),Person("Mary",28))
s match {
    case Son(Person("Jack",_),Person(motherName,_)) =>
          println("This is Jack's son. His mother "+motherName+".")
    case _ =>
}
```

But how does it work? To understand this, we need to first understand extractors.

An extractor is an object in Scala that has a method called *unapply*. When an extractor is used as a pattern, Scala will call its *unapply* method on the matching object. If the object can not be cast to the type of the parameter of the unapply method, then the matching fails immediately. If it can, then it will be passed as the parameter of the unapply method. The return type of the unapply method is usually an *Option* type. When the return value is a *Some* value, then it is matched, otherwise, if it is *None*, then matching fails.

```
object Twice{
     def unapply(s: String):Option[String] = {
         val half = s.subString(0,s.length/2)
         if(half == s.subString(s.length/2)) Some(half) else None
     }
}
val s:Any ="okok"
s match{
    case Twice(half) => println(half)
    case _ =>
}
```

The above code will print "ok". The *unapply* method of Twice will be called on *s*. If *s* is not a string, the first patten will fail. If s is not a concatenation of two same strings, then the it unapply method of Twice will return *None*, and the pattern will fail.For the above code,it will return *Some("ok")*, and *"ok"* will passed to parameter *half*.

If a class is declared as a case class, the Scala compiler will generate a companion object for it, with unapply method defined. So case class can be used directly as an extractor.

CHAPTER 3

Pattern-driven Programming

3.1 Definition

In pattern-driven computation, an instruction is executed only when a corresponding pattern matches, which is to use pattern to drive the control. Pattern-Driven programming is the combination of this type of control mechanism with either a shared memory or message passing data mechanism (PASH or PAME, respectively).

In Pattern-driven programming, we only define the different patterns that the arguments or context might match, and the corresponding actions to take if the pattern matches. So the control flow of the program totally relies on pattern matching.

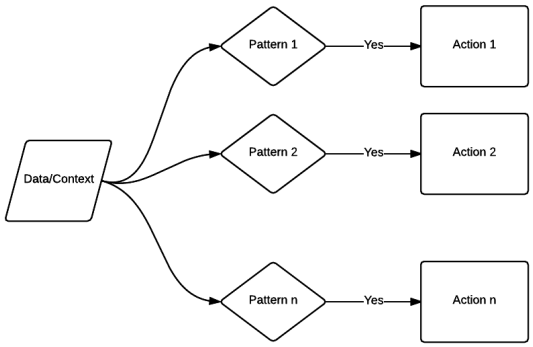

Figure 2: Independent pattern-driven programming paradigm

Based on the relationship between different patterns, we can divide pattern-driven programming into two categories, independent pattern-driven programming and dependent pattern-driven programming.

As shown in Figure 2, with independent pattern-driven programming, any pattern that matches will trigger the corresponding action immediately, without considering the results of other patterns. This computation model is inherently parallel,

since every pattern matching operation and its corresponding action is independent with each other, and there is no need for variable sharing and execution synchronization. The resolution problem of Prolog interpreter is good fit for this computation model. In Prolog, there is no order of rules. Every rule that matches is a valid solution.

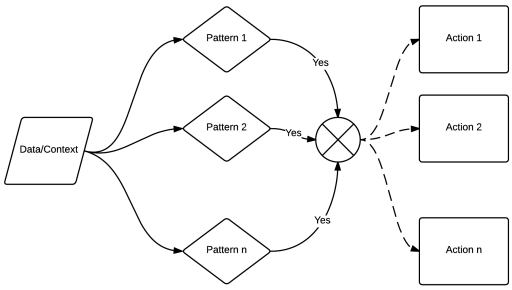

Figure 3: Dependent pattern-driven programming paradigm

Dependent pattern-driven programming is illustrated in Figure 3. When any pattern matches, it needs to consider other pattern matching results to decide whether the corresponding action should be executed. As shown in Figure 3, after every pattern is evaluated, there will be an operation to make a decision whether its corresponding action will be taken or not. That decision will consider the results of other patterns. That is the key difference with independent pattern-driven computation.

The dependencies between patterns could be in any form. The most common form is the order dependency, such as the first-match policy that the match expression in Scala uses. With the order dependency, each rule has a unique order, and the actions that could be triggered are decided on the order of the corresponding pattern. For example, the match expression will only trigger one action, which is the action corresponding to the first matched pattern.

3.2 PA-Scala

To explore the advantages and drawbacks of pattern-driven programming, we constructed a pure pattern-driven programming language by using a subset of Scala as our programming language, which we call PA-Scala (Pattern-driven Scala). To make PA-Scala a pure pattern-driven programming language, in which pattern-matching is the only way to drive the execution, we make the restrictions that only function call and pattern matching are allowed for controlling executions. We also define PA-Scala as a pure functional language. The functional features in Scala are kept, but mutable variables are not allowed in PA-Scala.

Sequential blocks, IF expressions and WHILE/FOR loops are not available in PA-Scala. The programs are composed of function calls, and in each function there's only pattern-matching. Using pattern-matching, another function call is determined to execute.

A typical function definition in PA-Scala looks like the following:

```
def myFunc(a:A,b:B):C = {
    t(a,b) match{
        case pattern1 => func1()
        case pattern2 => func2()
        . . . .
        case patternN =>funcN()
    }
}
```

There are three parts in each function:

1. Context transformation.

   Like the function call t(a,b) in the previous example, context is transformed by calling functions so as to form the data to be matched. The result of the

transformation appears on the left of the key word *match*.

2. Patterns.

   These are the patterns that the context could possibly match. Patterns appear after each *case* keyword. In the previous example, they are pattern1, pattern2, . . ., patternN.

3. Actions.

   Actions are function calls that followed each pattern after => symbol. In the previous example, they are func1(), func2(), . . . ,funcN().

Actions could also be inline pattern matching, which is just put the content of a function call directly after => symbol. So nested *match* expressions is also allowed:

```
def myFunc(a:A,b:B):C = {
    a match{
        case pattern1 => func1()
        case pattern2 => b match {
                case patternX => funcX()
                case patternY => funcY()
            }
    }
}
```

Because the *match* expression in Scala uses first-match policy, using PA-Scala we can only do dependent pattern-driven programming. The independent pattern-driven programming requires us to have a programming language and compiler that could do pattern matching in parallel and execute corresponding actions without considering the results of other pattern. So our exploration of pattern-driven programming will be focused on dependent pattern-driven programming, and we will leave the exploration of independent pattern-driven programming as future work.

3.3 Scala to PA-Scala

Even though PA-Scala does not have control structures like sequential execution, IF/ELSE, WHILE loop and FOR expression, we will show that PA-Scala can implement the same functionality easily, which means that every Scala program can be transformed to PA-Scala program. We use the procedure similar to small step semantics to illustrate how Scala control structures can be transformed to PA-Scala program. Small step semantics formally describe how the individual steps of a computation take place. Similarly, we describe how the individual steps of transformation is done to transform from Scala to PA-Scala.

- Sequential Blocks

  Sequential Blocks means that two or more statements execute in order, which means that the control is implicitly delivered from the first statement to the second statement. It is a control structure of command flow. In pattern-driven programming it is not allowed, since the only way to determine the next instruction is through pattern matching.
  One possible sequential blocks in Scala is like the following:

  ```
  e1
  e2
  ```

  where e1 and e2 are two expressions that execute one by one, and e2 does not rely on the result of e1, so in PA-Scala we can implement it as the following:

  ```
  e1 match {
      case _ => e2
  }
  ```

  If e2 relies on the result of e1, for example, e1 is an assignment expression:

```
val v = e1
e2(v)
```

then in PA-Scala, we can easily use the following way:

```
e1 match {
    case v => e2(v)
}
```

- IF/ELSE

  IF/ELSE replies on the result of the condition to jump to the corresponding branch. In Scala it is like the following:

```
if (e1)
    e2
else
    e3
```

  In PA-Scala it can be directly mapped to two patterns for the result of e1:

```
e1 match {
    case true => e2
    case false => e3
}
```

- WHILE loop

  In Scala a while loop will repeat the execution of the code block while the condition holds true:

```
while (e1)
    e2
```

  In PA-Scala, we can use recursion to achieve the same effect. We define function $f1$ as the following:

```
def f1() ={
    e1 match{
        case true => e2 match{
            case _ => f1()
        }
        case false =>
    }
}
```

So when $e1$ is evaluated to true, $e2$ will execute and then $f1$ will be called recursively. When $e1$ is evaluated to false, then the execution ends. Then the WHILE statement in Scala will be transformed to a function call f1() in PA-Scala

- FOR expression

  For expression in Scala looks like the following:

  ```
  for( x <- e1 if e2) yield e3
  ```

  Every for expression can be expressed in terms of three high-order functions: map, $flatMap$ and $withFilter$, and actually used by the Scala compiler [8].

  ```
  for( x <- e1) yield e2
  ```

  is translated to:

  ```
  e1.map( x=> e2)
  ```

  And the following for expression:

  ```
  for( x <- e1 if e2) yield e3
  ```

  is translated to:

```
e1 withFilter (x => e2) map (x => e3)
```

So we can see that the control structures that are missing in PA-Scala could easily be achieved by using pattern-driven programming, which means that it is possible to use only pattern matching and function call to drive the execution. And we can also see that the code size and running efficiency are basically the same.

CHAPTER 4

Phrase Structure Grammar Solver

Our first exploration of pattern-driven programming is using PA-Scala to implement a solver for phrase structure grammar. The solver's goal is to decide whether a given a string belongs to a language defined by a phrase structure grammar or not. There are two reasons that we want to implement a phrase structure grammar solver first:

1. Phrase structure grammar is equivalent to Turing machine in computability power [9], which means that if PA-Scala can solve the phrase structure grammar problem, we can prove that PA-Scala is computationally complete.

2. Phrase structure is a modification of a rewriting system [15], so the solver for phrase structure grammar will involve matching rewriting rules, where pattern matching can play an important role.

4.1 Phrase Structure Grammar

A phrase structure grammar [7] is a quadruple $G = (V, \Sigma, P, S)$, where

- V is a finite set of symbols called the vocabulary (or set of grammar symbols);
- $\Sigma \subseteq V$ is the set of terminal symbols (terminals);
- $S \in (V - \Sigma)$ is a designated symbol called the start symbol;
- $P \subseteq V^*NV^* \times V^*$ is a finite set of productions, where $N = V - \Sigma$ is called the set of nonterminal symbols.

Every production $\langle \alpha, \beta \rangle$ is also denoted as $\alpha \to \beta$. A production of the form $\alpha \to \epsilon$ is called an epsilon rule, or null rule.

Phrase structure grammars or type 0 grammars are the most general grammar in the hierarchy that Chomsky introduced [11], which are equal to the family of recursively enumerable languages. Recursively enumerable languages can be defined by Turing machines. So phrase structure grammar is equivalent with Turing machine in computability, which means that any problem that a Turing machine can solve could be mapped to a question whether a string belongs to a language defined by a phrase structure grammar. If PA-Scala could solve that problem, which is to use PA-Scala to write a phrase structure solver for that, then PA-Scala can be used for general programming problems.

4.2 Solver Implementation with Pattern-driven Programming

To determine whether a string belongs to the language $L(G)$ defined by a phrase structure grammer $G = (V, \Sigma, P, S)$, the solution that we applied is to try to reverse the string by applying the reverse rules of P in any possible way, and to find if there is one way that could reverse the string back to the start symbol S. If there is, then we can say that the string belongs to $L(G)$. If the string does not belong to $L(G)$, the procedure will either finish with no valid reverse to S found, or stuck in infinite search.

An example is shown in Figure 4, which is to determine whether string "CAAA" belongs to the language defined by a structure grammar G, where the rules P are the

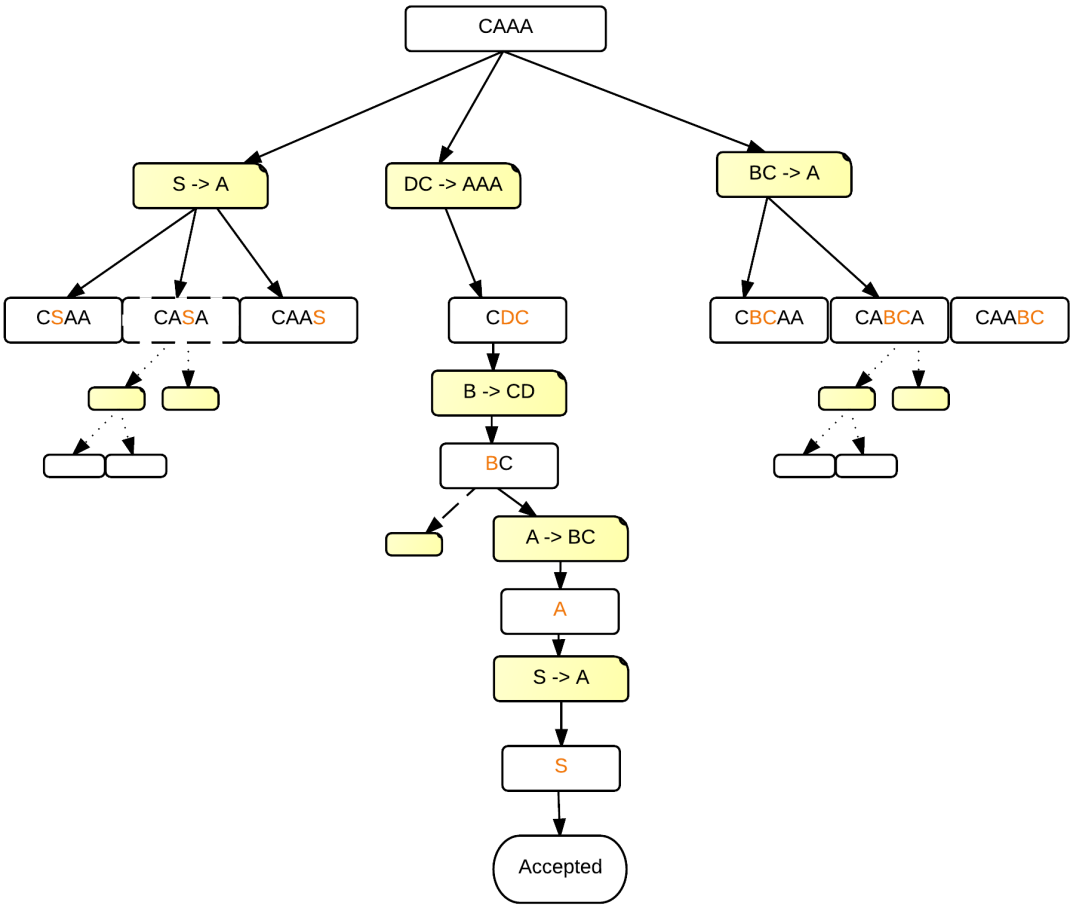

Figure 4: An example for solving phrase structure grammar

following:

$$S \to A$$

$$A \to BC$$

$$BC \to A$$

$$A \to DD$$

$$B \to CD$$

$$BB \to B$$

$$CD \to B$$

$$DC \to AAA$$

We take the initial string "CAAA" as the goal g_0, then for every rule $\alpha \to \beta$ we check whether g_0 is the form of $U\beta V$, where U is the prefix and V is the suffix. For every possible $U\beta V$, we substitute it with $U\alpha V$ to get a new goal g_i. Because any $U\alpha V$ could get $U\beta V$ by applying the rule $\alpha \to \beta$, then if the new goal (string) g_i belongs to the language of G, we know that g_0 belongs to the language of G. So we repeat the procedure on every new goal g_i, which will give us a tree structure like Figure 4. For each goal node (white node) in the tree, there could be zero to many rules that could be applied for reverse, so it will have zero to many rule nodes (yellow nodes) as its children. And for each rule node, it could have one to many ways to apply the reverse, based on the position of the reverse replacement in the goal, so it could have one to many new strings as the new goals, which are the goal nodes as the children. From Figure 4, we can see that "CAAA" could reduce to S by the following path:

$$CAAA \to CDC \to BC \to A \to S$$

which means that from the start symbol S we could get "CAAA" using rules in P, so "CAAA"is a string in the language of G.

Other than the path that reduced to S, the following reverse path is an infinite path by applying the reverse of the rule $BB \to B$ infinitely:

$$CAAA \to CDC \to BC \to BBBC \to BBBBC \to B \cdots BBBBC$$

To avoid running into this kind of infinite path before we find a finite path that could reduce to S, which will lead to infinite search even when there is a solution, a BFS (Breadth-first Search) strategy must be applied. So a path with smaller depth will always be explored before a path with the larger depth. Therefore if there is a finite path that could lead to S, it will always be explored before we run into any infinite

path.

We implemented a phrase structure grammar solver in PA-Scala, as shown in List 4.1, which includes four functions.

Listing 4.1: Phrase Structure Grammar Solver in PA-Scala.

```
1  object PAPSGSolver extends App {
2    case class Rule(condition:String,conclusion:String)
3
4    def solvePSG(ruleList:List[(String,String)],goal:String):Boolean = {...}
5
6    def solve(rules:List[Rule],allRules:List[Rule],goal:String,
7                        otherGoals:Queue[String],visited:Set[String]) :Boolean={...}
8    def tryRule(rule:Rule,goal:String) :(Boolean,Set[String])={...}
9
10   def matchRule(rule:Rule,goal:String,index:Int):(Option[String]) = {...}
11 }
```

The *solvePSG* function is shown in List 4.2. It takes a rule list and a goal string as input. The rule list is the representation of the rules of a phrase structure grammar, and the goal string is the string that we want to test. It will return a boolean value indicating whether the string can be constructed by the rule list or not, which is whether the string is in the language defined by the phrase structure grammar. A rule is composed of two parts: condition and conclusion. So a rule is represented by a tuple (String,String), where the first item is the condition, and the second item is the conclusion.

As we can see, there is just one match expression in that function. The object that is matched on is a tuple of the rule list and the goal. We can clearly see from

Listing 4.2: solvePSG Function in Phrase Structure Grammar Solver

```
1  def solvePSG(ruleList:List[(String,String)],goal:String):Boolean = {
2    (ruleList.map{x => Rule(x._1,x._2)},goal) match {
3      case (_,"S") => true
4      case (Nil,_) =>false
5      case (rules,_) => solve(rules,rules,goal,Queue(),Set())
6    }
7  }
```

the *match* expression that :

- If the tuple matches the pattern that goal is S, which means the goal is the start symbol, it will return *true*.
- If the tuple matches the pattern that the rule list is empty, considering the previous pattern is not matched, it will return $false$ because there is no way to construct the goal with empty rules.
- If none of the above patterns matches, it will extract the rule list out of the tuple, and try to solve it using the rules by calling *solve* function.

We can see that one benefit of pattern-driven programming is that an function can be easily interpreted by the reader as a list of options, because there is only pattern matching inside. And for every option, we can easily distinguish the condition (pattern) and the corresponding action. This makes the code very clear, which greatly improves the code readability.

In List 4.3 the *solve* function used nested *match* expressions to make further decisions. If nested *match* expressions are not used, the further decisions should be wrapped in other functions, which will lead to very scattered code that is hard to read and maintain. With two layers of nested *match* expression, the readability of

the code is still good, though not as good as the one layer *match* expression. But we tell that with too many layers of nested *match* expression, it will lose the benefit of the clarity of the code, and become even more difficult to read than conventional command-driven code, because the reader need to memorise and analyze the patterns that matched and not matched along the way to understand the conditions. So many layers of nested *match* expressions should be avoided in pattern-driven programming.

The *otherGoals* argument in function *solve* is a queue which helps to organize the search operation as a breadth-first search. The *visited* argument is a set of strings which keeps a record of the intermediate goals that have been added to the search queue, so that the same string will not be searched again in other branches. This will prevent duplicated searches, and also serves as a loop detector to prevent loop search.

In line 5 of code 4.3, that single *case* clause achieved several things:

- Makes sure that the queue *otherGoals* is not empty.
- Dequeues the head of the queue and assigned it to *newGoal*.
- Assigns the rest of the queue to a new variable *restGoals*.

This is the strength of pattern matching in Scala where pattern matching can make the code concise. By heavily relying on the pattern matching features of Scala, PA-Scala is also very concise and expressive.

Listing 4.3: solve Functon in Phrase Structure Grammar Solver

```
1  def solve(rules:List[Rule],allRules:List[Rule],goal:String,
2                  otherGoals:Queue[String],visited:Set[String]) :Boolean={
3  rules match{
4    case Nil => otherGoals match{
```

```
5           case newGoal +: restGoals => solve(allRules,allRules,newGoal,restGoals,visited+goal)
6           case _ => false
7         }
8         case rule :: restRules => tryRule(rule,goal) match{
9           case (true,_) => true
10          case (false,newSet) =>
11              solve(restRules,allRules,goal,otherGoals.enqueue(newSet.diff(visited)),visited)
12        }
13     }
14   }
```

Function *tryRule* in code 4.4 tests whether a rule can be used to reverse a goal. If it can, it will also return the possible reverse results, which could be used as new goals. In line 2 of the function, chained function calls of map,$filterNot$ replaces the for expression in Scala to iterate over all the possible positions to do the reverse. We can see that PA-Scala can support complex operations at transformation stage to prepare the object that will be matched on.

Listing 4.4: tryRule Functon in Phrase Structure Grammar Solver

```
1   def tryRule(rule:Rule,goal:String) :(Boolean,Set[String])={
2     (0 to (goal.length()−1)).map(matchRule(rule,goal,_)).filterNot(_.isEmpty).map(_.get).toSet match{
3       case set if set.contains("S") => (true,set)
4       case set => (false,set)
5     }
6   }
```

Function *matchRule* as shown in code 4.5 provides the function to reverse a goal by applying a rule. We can see that PA-Scala forces us to wrap this block of code to

a separate functions. The lack of the other control structures enforces the modularity of the code.

Listing 4.5: matchRule Functon in Phrase Structure Grammar Solver

```
1   def matchRule(rule:Rule,goal:String,index:Int):(Option[String]) = {
2    goal.indexOf(rule.conclusion,index) match{
3      case −1 => None
4      case i => Some(goal.substring(0, i) + rule.condition + goal.substring(i+rule.conclusion.length()))
5    }
6  }
```

In the code, we can see that the structure of pattern driven programming forces us to divide the code into small separate functions to avoid deep nested match expressions. Because one layer of match expression will just make one decision, so tasks with many decisions will have to be divided in to several small functions, otherwise there will be too many layers of pattern matching in a single function, which will make it hard to track the problem and not easy to understand. By encouraging small functions, it enforces the cohesion of every function, which makes the code much more modular and much easier to understand and maintain. The pattern-driven style of PA-Scala is straightforward and easy to be mapped from the problem that deals with different branches. But the drawback is that when there are very few branches or no branches (consecutive execution), the enforcement of small functions will make the code scattered and tedious, and the improvement on readability is trivial, and sometimes it will even hurt the readability.

CHAPTER 5

Prolog Interpreter

Prolog is a famous logic programming language, with declarative programming style. The Prolog program is expressed by defining facts and rules [14], and the computation is a process of querying all the constraints to find all solutions to a goal. The two major computation processes of the solver (Prolog interpreter) are resolution and unification. Resolution involves query through all the constraints to find the facts or rules that match the items in the goal, and unification tries to make two items that almost match to perfectly match by making variable substitution or value assignment. So the interpreter relies heavily on pattern matching. Therefore we think that the pattern-driven programming paradigm can be a great fit for writing the interpreter, not only by making the task much easier to model and implement, but also by making the code more elegant and much easier to understand and maintain.

To solve the problem step by step, we first implement an interpreter for Proplog, a small subset of Prolog, which is a logic language based on propositional logic. Then based on that, we implement the interpreter for Datalog, a language for predicate logic, which adds variables to Proplog. At last, we add data structure support to Datalog to get the interpreter of Prolog.

5.1 Proplog Interpreter

5.1.1 Proplog

Proplog [12] is a small subset of Prolog. The programs of prolog express that some proposition holds conditionally or unconditionally. A proposition is just a sentence. If a proposition is true when some other propositions are true, then we say that

proposition holds conditionally, and we call it a rule. Otherwise, if the proposition is true unconditionally, then we call it a fact. Actually, a fact can be thought of as a rule with no condition. A Proplog program is a database of facts and rules. A Proplog rule is of the form such as:

$$p :- q, r.$$

Where p is the conclusion,q and r following the symbol ":-" are the conditions. Conclusion could only have one proposition, where conditions could have many.

Then a fact of Proplog is a rule without condition, which is of the form:

$$p.$$

5.1.2 Proplog Interpreter With Pattern-driven Programming

We take a top-down approach to execute Proplog program. Starting from a given goal, we query the database of the rules and facts to find the rule or fact that could establish the first proposition in the goal list. Then we take the conditions of the rule or fact as sub goals to substitute that proposition in the goal list. We continue this process, and if finally all the propositions in the goal are facts, then the goal is established. If at last there are still propositions in the goal list that we could not find any substitution of facts, then the goal is not satisfied by the currently applied rules, then we will use backtracking to apply other rules. If the goal is not satisfied by any applied rules, then the goal is not established.

A proposition is just a string. A rule is a list of string, with its head as the conclusion, and its tail as the conditions. To implement this interpreter using PA-Scala, a rule can be represented with an instance of List[String], the program of

Listing 5.1: Proplog interpreter in PA-Scala.

```
def solve(goals:List[String],program:List[List[String]]) : Boolean = {
    goals match{
      case Nil => true
      case goal::remainGoals => program.exists(solvableWithRuleSubstitution(_,goal,remainGoals,program))
    }
  }

def solvableWithRuleSubstitution(rule:List[String],goal:String,remainGoals:List[String],
                                 program:List[List[String]]): Boolean={
    rule match{
      case Nil => false
      case `goal`::conditions => solve(conditions ::: remainGoals,program)
      case _ => false
    }
  }
```

Proplog can be represented with List[List[String]], and the goals are represented with List[String]. So the interpreter is a function that giving program as List[List[String]] and goals as List[String], to return a Boolean value to indicate whether the goals could be deduced from the program.

List 5.1 shows the only two functions in our Proplog interpreter implemented in PA-Scala, and each function is composed by just one match expression.

In PA-Scala program, because each function is just a match expression, it's very easy to read and understand what the code does. For example, in the *solve* function, there's only two cases: if *goals* is an empty list, return true; if *goals* is a list with *goal* as the head and *remainGoals* as the tail, then return whether a rule exists in the program that could substitute the *goal* to make the *goals* solvable. And we can see that it is not only simple and clear, but it is also just the direct translation of the literal expression.

In this program, pattern matching simplifies the code in the following ways:

1. Decomposed list to head and tail by pattern matching.

   In function *solve*, by matching *goals* to the case of $goal :: remainGoals$, we not only get the branch that *goals* has at least one element, but also defined variable *goal* and assigned $goals.head$ to it, and defined variable $remainGoals$ and assigned $goals.tail$ to it. All these are done by a single case expression, which will usually take several lines of code without pattern-matching.

2. Combined value comparison and variable list decomposition together by pattern matching.

   In function $solvableWithRuleSubstitution$ the case of '*goal*' :: *conditions* will make sure that rule has at least one element, and the head of rule equals to variable *goal*, and it also extract the tail of rule as conditions.

Using immutable variables, we also avoided the needs to manually make backtracking. As everything is immutable, we don't need to restore the state to start a new search.

To show the readability and clarity achieved by using PA-Scala, let's compare with the pseudo-code in List 5.2 for Proplog solver without pattern matching [12].

We can see that the pseudo-code is not only much more verbose, but also hard to get a clear structure out of it, therefore need more efforts to analyze and understand.

On the contrary, PA-Scala has a very clear structure, since every function follows the same style, which is just a match expression. So the reader knows the structure without even looking at the code. With the match expression structure in mind, the reader can quickly look through all the cases to get an idea all the branches the function has, then the reader can follow any interesting case to find out what the outcome for that would be. Because it just lists out all the possible cases and

Listing 5.2: Pseudo-code without pattern-matching

```
1  function establishtd(GOALLIST:symbollist): Boolean
2          var I: integer
3          begin
4              if GOALLIST = nil then return(true)
5          else
6              for I:=1 to MAXCLAUSE do
7                  if CLAUSE[I].HEAD = GOALLIST↑.SYM then
8                      if establish(copycat(CLAUSE[I].BODY),GOALLIST↑.REST)) then
9                          return(true)
10         return(false)
11 end;
12
13 function copycat(FIRST,SECOND:symbollist) :symbolist;
14         var COPYLAST:symbollist;
15         begin
16             if FIRST = NIL then copycat := SECOND
17             else
18                 begin
19                     new(COPYCATS)
20                     copycat := COPYLAST;
21                     COPYLAST↑.SYM := FIRST↑.SYM
22                     while FIRST↑.REST <> nil do
23                         begin
24                             new(COPYLAST↑.REST);
25                             COPYCATS := COPYLAST↑.REST;
26                             FIRST := FIRST↑.REST;
27                             COPYLAST↑.SYM := FIRST↑.SYM
28                         end;
29                     COPYLAST↑.REST := SECOND
30                 end;
31         end;
```

the corresponding consequence, it presents a one layer flat structure, which is very straightforward and very similar to the way that human thinks, and therefore very easy to understand and reason about.

5.2 Datalog Interpreter

5.2.1 Datalog

Datalog is a subset of Prolog, and a superset of Proplog. Datalog provides parameterized version of Proplog clauses, which makes it more expressive. It also provides variable support, so Datalog interpreter can produce values as answers to a query. But different from Proplog, Datalog will try to find all the possible values that satisfy the query as the answer, which means that there could be multiple answers for a query.

A rule of Datalog can be a parameterized Proplog rule, which is in the form such as:

$$p(X,Y) :\!\!- \; q(X),\; r(X,Y).$$

where $p(X, Y)$ is the conclusion, $q(X)$ and $r(X, Y)$ are conditions. There could be many conditions, but still only one conclusion.

In Datalog, a parameterized proposition is called a literal [12], which is in the form of $proposition(paramList)$. A key operation in Datalog interpreter is to match literal against another to determine the minimum amount of filling or substitution to make them match perfectly so as to satisfy a sub-goal for a clause. This operation is called unification.

Definition [13] A *substitution* θ is a finite set of the form $\{v_1/t_1, ..., v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i, and all variables v_i are distinct. If E is an expression, then $E\theta$ is an expression obtained from E by replacing each occurrence of the variable v_i in E by the corresponding terms t_i .

Definition [13] For a finite set of literals S, if a substitution A could make

all the literals the same (singleton), then substitution A is called a *unifier* for S. The minimum amount of substitution to unify S that we are looking for is called a $mostgeneralunifier$ (MGU) for S [13]. Any unifier of S could be acquired by applying more substitutions to the MGU of S. In other words, if A is a unifier of S, and for any unifier B of S, we can find a substitution θ so that $B = A\theta$, then A is called a MGU of S.

It is proved in [13] that if we use the following unification algorithm, we will get the MGU of set S if S is unifiable:

UNIFICATION ALGORITHM [13]

1. Put $k = 0$ and $\sigma_0 = \epsilon$.
2. If $S\sigma_k$ is a singleton, then stop; σ_k is an mgu of S. Otherwise, find the disagreement set D_k of $S\sigma_k$
3. If there exists v and t in D_k such that v is a variable that does not occur in t, then put $\sigma_{k+1} = \sigma_k v/t$, increment k and go to 2. Otherwise, stop; S is not unifiable.

This unification algorithm is non-deterministic, because there could be several choices in step 3 for v and t, but for any two MGU produced by this algorithm will only differ by a change of variable names.

5.2.2 Datalog interpreter in PA-Scala

We can define literals and propositions as a unique Predicate class:

```
case class Predicate(name:String,paramNum:Int,params:List[String])
```

So literals are instances of *Predicate* where *paramNum > 0* and *params* is not empty, while propositions are instances of *Predicate* where *paramNum = 0* and *params* is empty. For example, literal *A(X,Y)* is an instance of *Predicate* as *Predicate("A",2,List["X","Y"])*, while proposition *B* is an instance of *Predicate* as *Predicate("B",0,Nil)*. So a clause in Datalog is an instance of *List[Predicate]*, and the program is of type *List[List[Predicate]]*. The goals are conversed to an instance of *List[Predicate]*. A unification or unifier is of type *Map[String,String]*, where the key is the original symbol, and the value is the substitution.

The unification operation is the most important part of a Datalog interpreter, and also the part that replies heavily on pattern matching. So we will just show the two functions that are used to do unification.

Listing 5.3: Predicate Unification in PA-Scala

```
1  def matchPredicate(goal:Predicate,restGoals:List[Predicate],oldUnification:Map[String,String],
2                     rule:List[Predicate]) :(Boolean, Map[String,String],List[Predicate]) = {
3    instantiate(rule) match{
4      case Predicate(‘goal‘.name,‘goal‘.paramNum,params)::conditions =>
5           matchParams(goal.params,params,oldUnification) match{
6                case (true,_,_,unification) =>
7                     (true,unification,substitute(conditions ++ restGoals, unification))
8                case _ => (false,Map[String,String](),Nil)
9      }
10     case _ => (false,Map[String,String](),Nil)
11   }
12 }
```

Function *matchPredicate* in code 5.3 try to unify a goal with the conclusion (head) of a rule. If the conclusion has the same name and same number of parameters with the goal, then we further unify them by matching the two parameter lists by calling function *matchParams*, which is shown in code 5.4. Otherwise, the goal and the conclusion of the rule could not be unified. By unifying the goal and the conclusion of the rule, we got a new unifier by adding the new substitutions to the old substitutions. We get the new goal list by appending the conditions of the rule to the remaining goal list. The new goal list should be updated by applying the new substitutions.

In line 4 of function *matchPredicate*, we uses a single case to describe the condition that the conclusion of the rule has the same name and the same *paramNum* with the goal, and also extracted the condition of the rule and the parameters of the conclusion of the rule.

Listing 5.4: Parameters Unification in PA-Scala

```
1  def matchParams(goalParams:List[String],ruleParams:List[String],
2                  oldUnification:Map[String,String]):(Boolean,List[String],List[String],Map[String,String]) = {
3          (goalParams,ruleParams) match {
4          case (Nil,Nil) => (true,Nil,Nil,oldUnification)
5          case (x::xs,y::ys) if x == y => matchParams(xs,ys,oldUnification)
6          case (x::xs,y::ys) if isVariable(y) =>
7                  matchParams(substituteParams(y,x,xs),
8                                  substituteParams(y,x,ys),
9                                  substituteUnification(y,x,oldUnification)+(y->x))
10         case (x::xs,y::ys) if isVariable(x) =>
11                 matchParams(substituteParams(x,y,xs),
12                                 substituteParams(x,y,ys),
13                                 substituteUnification(x,y,oldUnification)+(x->y))
```

```
14          case _ => (false,Nil,Nil,Map[String,String]())
15          }
16  }
```

Function *matchParams* uses pattern matching to describe all the possibilities regarding matching two parameter lists.

1. If the heads of the two parameter lists are exactly the same, we unify them by unifying the tails of the two parameter lists.

2. If the head of the second parameter list, y, is a variable, then we make a substitution as y/x, which is to substitute y with x, where x is the head of the first parameter list, no matter whether x is a variable or a constant. The substitution is applied to the remaining parameters and also the old unification. Then we recursively unify the tails of this two parameter lists.

3. If y is not a variable but x is a variable, then we make a substitution as x/y, which is to substitute x with constant y, and apply that to the remaining parameters and also the old unification. Then we recursively unify the tails of this two parameter lists.

4. Otherwise, the two parameter lists can not be unified, because neither x nor y is a variable, so they can not substitute with each other, and $x! = y$, so there's no way to make the heads of this two parameter list the same, therefore no way to unify the two parameter list.

By using PA-Scala to implement the interpreter in a pure pattern-driven style, we can see that matching between goals and rules is made really simple and straightforward. No explicit comparison operation is needed, and all the value comparisons

and value extractions are done in a single place, which makes the intent of the code extremely clear and makes both code writing and code reading easy and intuitive.

Also as we can see in function *matchParams*, the PA-Scala program is almost identical to the literal expression of the function's purpose. Writing the code is just mapping the different branches to different cases and nothing more. When reading the code, the reader could easily navigate into different branches by just take a glance on the cases without reading other parts of the code. We can see that PA-Scala has an advantage that the reader could get an idea of the main structure of a function by only looking a very small parts of the program (the cases), which will greatly help the process of code reviewing, debugging and maintaining.

5.3 Prolog Interpreter

5.3.1 Prolog

By introducing variables and allowing the program to give answers to queries, Datalog greatly improves Prolog's expressiveness and usefulness. Based on Datalog, Prolog adds complex data structures support, which allows the programmer to define and manipulate data structures (including nested data structures).

To extend Datalog into Prolog, a data structure facility, record, is introduced. A record is composed of a record name, followed by a list a values in the parentheses:

$$recordName(v_1, v_2, \cdots v_n)$$

The value in Prolog could also be a record. So a value in Prolog could be record, constant and variable. We use term to mean any value, whether it is record, constant, or variable. So a record has a record name, followed by a list of terms.

5.3.2 Prolog Interpreter with PA-Scala

The major difference between Prolog and Datalog is that in Datalog the parameters of a predicate are constants or variables, but in Prolog they could also be a record, which has its own data structure. So the major challenge for adding data structure feature to Datalog to get Prolog is to deal with record during unification for parameter lists. As predicate and record have the same data structure, we defined class Struct to present both predicate and record:

```
case class Struct(name:String,paramNum:Int,params:List[Term]) extends Term
```

We will only show the code for unification operation that how data structure or *Struct* is supported.

Listing 5.5: Struct Unification for Prolog Interpreter in PA-Scala

```
1 def matchStructs(goalStruct:Struct,ruleStruct:Struct,oldUnification:Map[String,Term]
2                                   ) :(Boolean,List[Term],List[Term],Map[String,Term]) ={
3   ruleStruct match{
4     case Struct(‘goalStruct‘.name,‘goalStruct‘.paramNum,params) =>
5               matchParams(goalStruct.params,params,oldUnification)
6     case _ => (false,Nil,Nil,Map[String,Term]())
7   }
8 }
```

Function *matchStructs* shown in List 5.5 tries to match two structs, which could be two predicates or two records. A pattern in line 4 makes sure that the two Struct instances have the same name and same parameter numbers, and extracted parameter list out and assigned to variable params. When this pattern is matched, function

matchParams is called to match and unify their parameter lists. Otherwise, these two predicates or records do not match at all and are not unifiable.

Listing 5.6: Parameters Unification for Prolog Interpreter in PA-Scala

```
1  def matchParams(goalParams:List[Term],ruleParams:List[Term],
2            oldUnification:Map[String,Term]):(Boolean,List[Term],List[Term],Map[String,Term]) = {
3    (goalParams,ruleParams) match {
4    case (Nil,Nil) => (true,Nil,Nil,oldUnification)
5    case (x::xs,y::ys) if x == y => matchParams(xs,ys,oldUnification)
6    case (x::xs,Variable(y)::ys) =>
7                              matchParams(substituteParams(y,x,xs),
8                                          substituteParams(y,x,ys),
9                                          substituteUnification(y,x,oldUnification)+(y->x))
10   case (Variable(x)::xs,y::ys) =>
11                             matchParams(substituteParams(x,y,xs),
12                                         substituteParams(x,y,ys),
13                                         substituteUnification(x,y,oldUnification)+(x->y))
14   case ((x:Struct) :: xs,(y:Struct) ::ys) => matchStructs(x,y,oldUnification) match {
15                             case (true,_,_,unification) =>
16                                  matchParams(xs.map(substituteTerm(_,unification)),
17                                              ys.map(substituteTerm(_,unification)),
18                                              unification)
19                             case (false,_,_,_) =>(false,Nil,Nil,Map[String,Term]())
20   }
21   case _ => (false,Nil,Nil,Map[String,Term]())
22   }
23 }
```

Function *matchParams* in Listing 5.6 tries to match and unify two different parameter lists. The function *matchParams* is very similar to the function *matchParams* in Datalog interpreter, except that it has to deal with the case that when both terms are records. And the way to deal with this issue is just to recursively call function *matchStructs*.

From the code above, we can clearly see the advantage of PA-Scala that the code is very easy to extend and maintain. We can easily extend the functionality of the code to support the structure feature by simply adding new patterns, without changing any structure of the code. Also the extended functionality stands out clearly when comparing with the original code, which easily shows the intent of the added code.

CHAPTER 6

Parallel Execution

As discussed in chapter 3, independent pattern-driven computation is inherently parallel, because the patterns can be evaluated simultaneously, and the corresponding actions could also run independently. For dependent pattern-driven computation, even though the actions are always dependent, the operations to test whether each pattern matches still can be executed in parallel. So it is possible to distribute tasks to multiple cores and let the pattern matching tests run simultaneously. In chapter 3 we already know that a PA-Scala program can get equivalent performance as a normal Scala program using single thread execution. So if PA-Scala could use parallel execution to speed up the execution, it means that we could utilize multiple cores to speed up a single thread task by writing that task in pattern-driven programming language like PA-Scala. This could give PA-Scala or pattern-driven programming an advantage to have the ability to naturally utilize multi-core processors to speed up execution for programs that could only be executed in a single thread in a traditional command-driven programming language like Scala and Java. In this section, we discuss the way to run PA-Scala programs in parallel execution, then we discuss the performance improvement that could be achieved by that, and discuss what kind of applications could get the most benefit from it.

6.1 Parallel execution for PA-Scala

The Scala compiler will compile PA-Scala code and execute the pattern matching operations sequentially as shown in Figure 5.

Suppose there are N patterns in a match expression, from pattern 1 to pattern

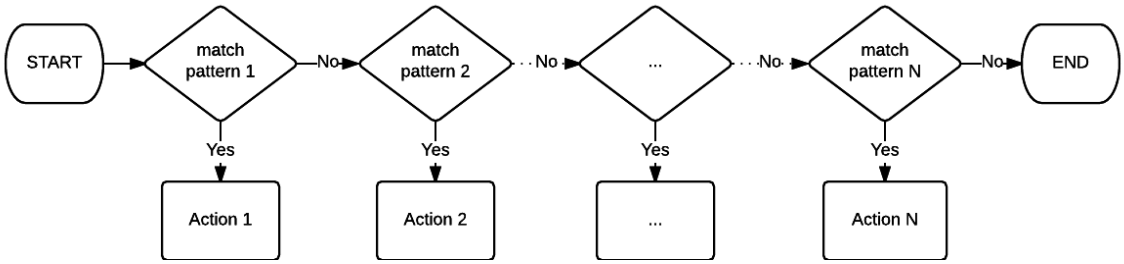

Figure 5: Sequential execution for pattern matching

N, then pattern 1 will be tested first to see whether it matches. If it matches, then the corresponding action will be triggered, and the pattern matching operation is finished. If it does not match, then the following pattern will be examined in the same way. This process will continue until no pattern is left to match. So in the worst case, all N patterns will be examined for pattern matching. The best case is that the first pattern matches because only one pattern need to be examined. Therefore the average case is to examine $\frac{1+N}{2}$ patterns sequentially.

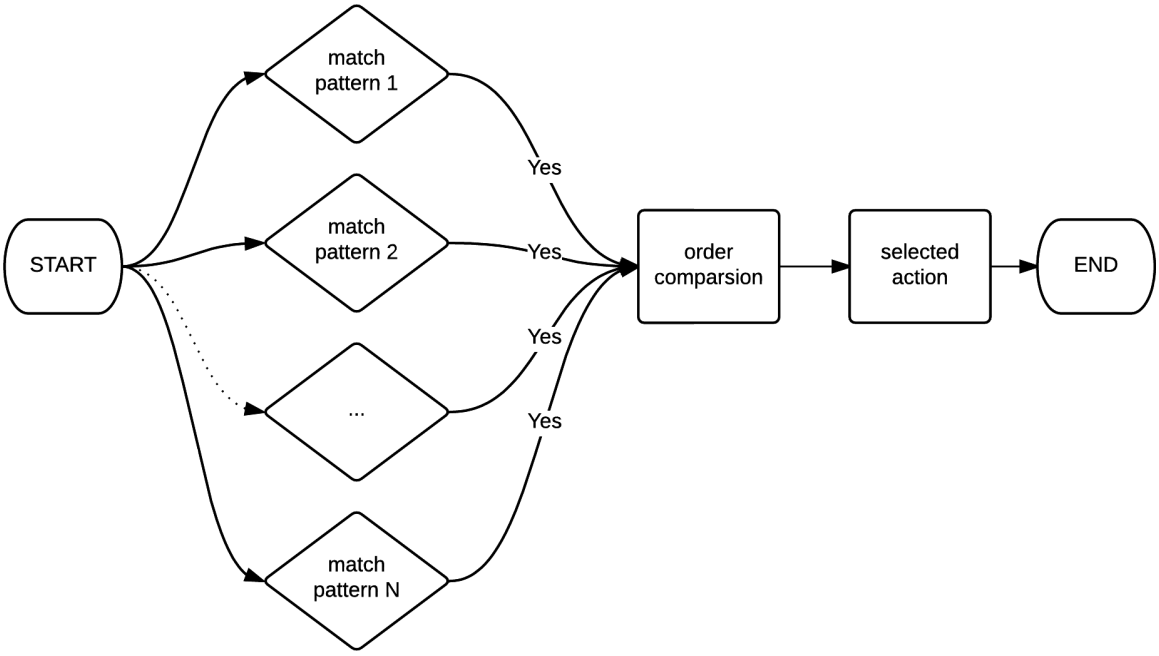

Figure 6: Parallel execution for pattern matching

But if there are multiple cores available, we can distribute each pattern test to

one core and run the all of the pattern matching tests simultaneously. Then based on the test results for all the patterns, we can decide which pattern is the first one matched, and therefore which action should be triggered, as illustrated in Figure 6. So if a complier complies the PA-Scala code to execute in this way, it could utilize the multiple cores on the modern computer to execute the pattern matching operations in parallel.

6.2 Efficiency Analysis

A typical function in PA-Scala looks like the following:

```
def f(a:A):B = {
    t(a) match {
        case p1 => action1()
        case p2 => action2()
        . . .
        case pN  => actionN()
     }
}
```

Three parts contribute to the running time of such a function::

1. Call the transformation functions to get the variables for pattern matching.

2. Do pattern matching and determine the action to take.

3. Call the corresponding action function.

These three steps will always be executed, and will always be executed one after another, no matter if we do pattern matching in parallel or not. Also the transform function and the triggered action function will remain the same. The only difference is how the pattern matching tests are performed.

To measure the running time of a program, we define the running time of the whole program as the sum of the running time of all the functions that are called. Suppose a program contains m functions, T_i denotes the running time of ith function, which was executed C_i times, then the running time of the whole program can be calculated as the following:

$$T_{program} = \sum_{i=1}^{m} T_i \times C_i$$

For example, if the whole program is just to call the typical PA-Scala function f, as shown above, and $actionN$ is the chosen action, then there are totally three functions called: f, transform function t and chosen action $actionN$, and they are all called only once. So the total running time of the program is:

$$T_{program} = T_f + T_t + T_{actionN}$$

where the running time of t and $actionN$ are not included in the running time of f.

In this way, we are using amortized running time analysis for each function to get the total running time. In a typical function f of PA-Scala, the running time is amortized in the following way:

1. The running time of the transform functions will be calculated separately. It should not be included in the running time of the function f itself to avoid duplicate calculation.Then the running time that f uses to do the transformation is just the running time to invoke the transform functions, which is constant. So $T_{call-transform} = O(1)$.

2. The running time for pattern matching is all counted as the running time of function f, which is $T_{pattern-matching}$.

3. The running time of the action function should also be calculated separately. So $T_{call-action} = O(1)$, since the running time to allocate and invoke the action function is constant.

We can think of the first function that the program called as an action function, too. So the running time of a program is the sum of the running time of all the transform functions and action functions that are executed.

The running time of a function is:

$$T_f = T_{call-transform} + T_{pattern-matching} + T_{call-action} = T_{pattern-matching} + O(1)$$

We can see that the running time of a function f in PA-Scala is totally dominated by the running time of the pattern matching operation, which is to find the pattern that first matches.

Suppose in function f, there are N patterns, and each pattern matching test costs the same amount of time, which is $O(T)$, then the average running time of the pattern matching in sequential execution is $O(\frac{(N+1)T}{2})$, which is $O(NT)$.

If the pattern matching tests are all executed in parallel, then $O(T)$ is needed to test all the patterns if we have N cores to use.Then to determine which matched pattern is the first in the order, the running time is at most $O(N)$. So the total running time of the pattern matching is $O(T + N)$.

When T is very small, there is no significant efficiency improvement by running the pattern matching in parallel. But if T is large, the parallel version is N times faster than the sequential version (assuming we have enough cores).

As mentioned in chapter 2, in Scala pattern matching is done by invoking the *unapply* method of an extractor. The *unapply* method is just another normal function,

which could also have many patterns to match. So the running time of *unapply* method could also be improved by running in parallel, which means that the running time for each pattern matching test, $O(T)$, could also be smaller. Suppose there are N' patterns in a *unapply* method in average, then running time of each pattern test will drop from $O(T)$ to $O(\frac{T}{N'})$ if the pattern matching in the *unapply* method was also run in parallel. Then the totally running time of the pattern matching operation will be $O(\frac{T}{N'} + N)$. Recursively, the pattern matching operations in the *unapply* method are also implemented by calling corresponding *unapply* method, which could also be improved by running in parallel. If we have enough cores to run pattern matching in every level in parallel, and suppose there are K levels pattern matching operations, where the average numbers of patterns for a pattern matching operation in each level are $\{N_1, N_2, \cdots, N_k\}$, then the total running time of the out most pattern matching operation is $O(\frac{T}{\prod_{i=2}^{k} N_i} + N_1)$. So when T is large and we have enough cores, the running time of a function for the parallel execution is only $\frac{1}{\prod_{i=1}^{k} N_i}$ of the running time for the sequential execution.

So parallel execution for pattern matching will achieve great performance improvement when the PA-Scala function is complex, which means that:

1. There are many patterns in the match expression.

2. Each pattern examination operation is time consuming, or the corresponding unapply method is also complex.

The gesture recognition problem is a good example . On touch screen devices like smart phones, gestures are widely used as user's inputs, such as flick, swipe, pinch, rotate, tap, double tap and drag. Based on a sequence of user's touch events, the system needs to identify which gesture it is, and also the attributes of the gesture,

such as the coordinate and the speed. Then the system will trigger the corresponding event handler for that gesture. If we define each gesture as a pattern by defining the corresponding extractor, then we can implement gesture recognition in PA-Scala like the following:

```
def onEvents(events:List[Event]) = {
    events match {
      case Flick(start,end,speed) => handleFlick(start,end,speed)
      case Swipe(start,end,speed) => handleSwipe(start,end,speed)
      case Drag(from,to)          => handleDrag(from,to)
      . . .
      case _                      => waitForMoreEvents(events)
    }
  }
object Flick{
    def unapply(events:List[Event]):Option[(Coordinate,Coordinate,Float)] = {
        events match {
            . . .
        }
    }
 }
object Swipe{ . . .}
. . .
```

In the *onEvents* function, the event sequence is matched to different gesture patterns to determine the gesture of the user and its corresponding attributes. It has a large number of patterns in the *match* expression since there are many possible gestures. At the same time the *unapply* method for each extractor involves a complex computation, which is to determine whether it is the corresponding gesture and the attributes for that gesture based on the event sequences. This is a typical scenario where we could get great performance improvement by using the parallel execution as we mentioned above.

We can see that pattern-driven programming paradigm has the potential to achieve performance improvements by running pattern matching tests in parallel. For problems that only involve simple pattern matching operations, the improvements might be trivial. But for problems that have large number of patterns, or complex and time consuming pattern matching operations, the performance improvements could be significant.

CHAPTER 7

Conclusion

The following are the contributions made by this thesis:

- Defined a pure pattern-driven language by using a subset of Scala, which made pattern-driven programming exploration possible and easy to do.
- Showed that pattern-driven programming can be used for general programming.
- Explored the strengths and limitations of pattern-driven programming by implementing a Prolog interpreter.
- Showed the potential for pattern-driven programming to improvement performance by parallel execution.

As any other programming paradigms, pattern-driven programming may not be the best fit for every problem. But as shown in this thesis, when the problem relies heavily on pattern matching operations, or the program could be abstracted and mapped to high level pattern matching problems, then pattern-driven programming could make the code intuitive and elegant, which makes the program not only easy to write, but also easy to understand and maintain. And due to the parallel nature of pattern matching operations, pattern-driven programs have the potential to utilize multiple cores on the modern computers to speed up executions for problems that can only be run sequentially in command-driven paradigm. So it could achieve great performance improvements for certain problems.

Even though we only explored dependent pattern-driven programming in this thesis, many features and principles should apply to independent pattern-driven pro-

gramming. To further explore independent pattern-driven programming, due to its first-match policy inherited from Scala, PA-Scala is not sufficient anymore. So to build a programming language and execution environment that could do independent pattern-driven programming and computation is our job in the future.

LIST OF REFERENCES

[1] J.Pearce, *Programming Language Concepts*,2015 `http://www.cs.sjsu.edu/faculty/pearce/modules/lectures/languages/Paradigms.htm`

[2] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins *Data-Driven and Demand-Driven Computer Architecture*, ACM Comput. Surv. 14, 1 (March 1982), 93-143.

[3] J.A. Sharp, *Data Flow Computing: Theory and Practice*, Ablex, 1992

[4] T. MOTO-OKA, *Fifth generation computer systems*,North-Holland, 1985

[5] D.H.D Warren, *A view of the fifth generation and its impact*,AI Magazine,Volume 3 Number 4, 1982

[6] J.B. Dennis, *Data flow supercomputers* Computer 11: 48-56,1980

[7] J. Gallier, *Notes on Formal Languages, Automata and Computability*, 2009, `http://www.cis.upenn.edu/~jean/gbooks/cis51104sl13pdf.pdf`

[8] M. Odersky, L. Spoon and B. Venners *Programming in Scala*, Second Edition, Artima, 2010

[9] J.E. Hopcroft and J.D. Ullman *Formal Languages And Their Relation to Automata*, 111-112, Addison-Wesley, 1969

[10] J.E. Hopcroft, R. Motwani and J.D. Ullman *Introduction to Automata Theory, Languages, and Computation*, Second Edition, Addison-Wesley, 2001

[11] N. Chomsky *On certain formal properties of grammars*, Information and Control, Vol. 1, pages 91-112, 1959

[12] D. Maier and D.S. Warren *Computing with Logic*, Benjamin/Cummings, 1988

[13] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984

[14] W.F. Clocksin, C.S. Mellish *Programming in Prolog*,Fifth Edition,Springer, 2003

[15] G. Gazdar, E.Klein, G. Pullum, I. SAG *Generalized Phase Structure Grammar*,Harvard University Press,1985

[16] *Lazy Evaluation*, 2015, Retrieved from `https://wiki.haskell.org/Lazy_evaluation`