L7: Records and Pattern Matching

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

January 28th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Components of PL

- Syntax: How do you write the language? {;}
- Semantics: What do program mean? What is the rule of evaluation
  - I.e., what are the evaluation rules?
- Idioms: What are the typical patterns for using language features to express computation?
- Libraries: What facilities does the language provides?
  - I/O, Data structures, etc.
- Tools: What is provided to make your job easier?
  - A debugger
  - REPL interface

Mutable vs. Immutable - Program is bug free (No program can modify)

- At this point you probably realize you can modify a list
  - You can append to existing list to create a new list
- What does this mean?
  - Let's say x is mapped to a value (which can be a List(1,2,3))
  - This x will be forever mapped to this list, and nothing will change x to map to a different list (immutable)
- Generally, we have a construct to build compound data and accessing pieces of compound data
  - But no construct to mutate the data we built

Mutable vs. Immutable

- Immutable benefits
  - You can guarantee no other code is doing something that make your code wrong (example: no one can modify existing lists)

- Example

Example

- def sortPair(p: (Int, Int)): (Int, Int) =
    if(p._1 < p._2) p else (p._2, p._1)
- def sortPair2 (p: (Int, Int)): (Int, Int) =
    if(p._1 < p._2) (p._1, p._2) else (p._2, p._1)

Same

- What are the differences between the two considering:
  - If a pair is immutable
  - If a pair is mutable

- For a language that allows mutable data, the two functions behave differently

Example #2

- def concat(xs: List[Int], ys: List[Int]): List[Int] =
  if (xs.isEmpty) ys else (xs.head)::concat(xs.tail, ys)
- Let’s assume xs = List(1,2) and ys = List(3,4,5)
- What can be the difference if we assume
  - Mutation is allowed
  - Mutation is not allowed

Generalizing Compound Types

- Product type: "each of" each value can be many types
  - A value contains values of predefined types
  - Example: Tuple

- Sum type: "One of" one of these types which are defined
  - A value is one of many types
  - Example: Option nothing and something (int, string)

- Recursive: Making self reference
  - A value of type T can refer to a value of type T
  - Example: List

Type Alias (nickname)

- You can define an alias of a type

- Example: Person consist of 4 different items
  - type Person = (String, Double, Int, String)

- This might still be annoying because you need to remember what values should go in which order
  - First entry in the tuple is the name
  - Second entry is the height
  - Third entry is the age
  - I cannot even come up with what should go into the fourth …

Record

- Record addresses the problem we just discussed
- case class Person(name: String, height: Double, age: Int, address: String)
- This make a named record for Person
- To use the record you make, you can:
  - Person("John", 1.80, 30, "Thailand") (Reference by Position)
    - Notice you need to have the correct order
  - Person(height=1.80, address="Thailand", age=30, name="John") (Reference by name) No need the order
- You can also bind a named record to a name using val
  - val p1 = Person("John", 1.80, 30, "Thailand")
- You can use the fieldname to access individual field
  - p1.name
  - p1.address

Reference by Name vs. Position

- Notice how you can refer to items in a record by name
- While you can refer to items in a tuple by position

- Different programming language can use either one, or a hybrid approach
  - Java method arguments
    - Caller uses position, callee uses variables
  - Python
    - By position for required arguments and by name for optional arguments

Ex. add(1,2) Position

add(int a1, int a2) Name

return a1+a2

Ex add(1,2 , type = "double")

Syntactic Sugar

- Basic idea: Making semantic easier to use

- Example: you can implement a tuple using records
  - case class MyPair(_1:Int, _2: Double)
- We will call this “tuples are syntactic sugar for records”

- Basically syntactic doesn’t introduce a new semantics
  - But repackage it to something that looks nicer

Creating Sum Types

- Let's expand our exposure to sum types beyond options
- What if we want to create all arithmetic expressions that involve addition and multiplication

Create new sum type call Expr

```
trait Expr
case class Constant(n: Double) extends Expr
case class Negate(e: Expr) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr
```

Creating Sum Types

- trait Expr
  case class Constant(n: Double) extends Expr
  case class Negate(e: Expr) extends Expr
  case class Sum(e1: Expr, e2: Expr) extends Expr
  case class Prod(e1: Expr, e2: Expr) extends Expr

- Expr is one of the following:
  - A constant with value n, type double
  - A sum of two expressions
  - A product of two expressions

Example

- What if I want to create a rank of playing cards
  - Jack, Queen, Ace, King and all the numbers

- trait Rank
  case object Jack extends Rank
  case object Queen extends Rank
  case object King extends Rank
  case object Ace extends Rank
  case class Num(num: Int) extends Rank

  Any number is possible

- Notice we mix up both class and object in this sum type

Pattern Matching with Sum Types

- As discussed previously, you can pattern match sum types
  - Example: pattern matching objects
- Let’s assume the following for our example

```
trait Expr
case class Constant(n: Double) extends Expr
case class Negate(e: Expr) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
case class Prod(e1: Expr, e2: Expr) extends Expr
```

Example 1

- What if we want to evaluate the sum type

- def eval(e: Expr): Double = e match {
  case Constant(n) => n input is constant, return n
  ~~case Negate(e) => - eval(e)~~ Don't need to write
  case Sum(e1, e2) => eval(e1) + eval(e2)
  case Prod(e1, e2) => eval(e1) * eval(e2) }

Example 2

- What about just printing the expression

- def stringify(e: Expr): String = e match {
  case Constant(n) => n.toString
  ~~case Negate(e) => "-" + stringify(-e)~~
  case Sum(e1,e2) => stringify(e1) + " + " + stringify(e2)
  case Prod(e1,e2) => "(" + stringify(e1) +")*(" + stringify(e2) + ")" }

String represent a+b

Example 1+2

- We can combine the two as one object
- Object ExprEval{

```
def eval(e: Expr): Double = e match {
case Constant(n) => n
case Negate(e) => - eval(e)
case Sum(e1, e2) => eval(e1) + eval(e2)
case Prod(e1, e2) => eval(e1) * eval(e2) }

def stringify(e: Expr): String = e match {
case Constant(n) => n.toString
case Negate(e) => “-” + stringify(-e)
case Sum(e1,e2) => stringify(e1) + " + " + stringify(e2)
case Prod(e1,e2) => "(" + stringify(e1) +")*(" +
stringify(e2) + ")" }
}
```

Default Case

- Similar to a switch case statement, we can have a default case

- case _ => [code goes here]

↳ Doesn't match to any cases, run this case

Pattern Matching Benefits

- Generally making codes look less ugly
- You get warning if you miss any cases
  - Or if you have duplicated cases (Negate)
- Works for both options and list

Before We Leave Today

In-class Exercise 7

- Implement the following function
- def zip(x : List[Int], y: List[Int]) : List[(Int, Int)] takes, for example, (List(3,2,5), List(6,1,9)) and returns List((3,6), (2,1), (5,9)).
  Hint: you can pattern match on tuples. case (Nil, Nil) is valid.

- def unzip(zipped : (List[Int], List[Int])) : (List[Int], List[Int]) takes, for example, (List((3, 6), (2,1), (5,9)) and returns (List(3, 2, 5), List(6, 1, 9)). s