PLRG

CS320 - Scala Seminar

Pattern Matching

Jaemin Hong

hjm0901@gmail.com

November 7, 2018

PLRG

Datatypes

- In many cases,
    - a single datatype contains multiple heterogenous shapes of values.

- For example, ...

Natural Numbers

- A natural number is
  - zero, or
  - a successor of a natural number.

- `0`, `1 = S(0)`, `2 = S(1) = S(S(0))`, ...

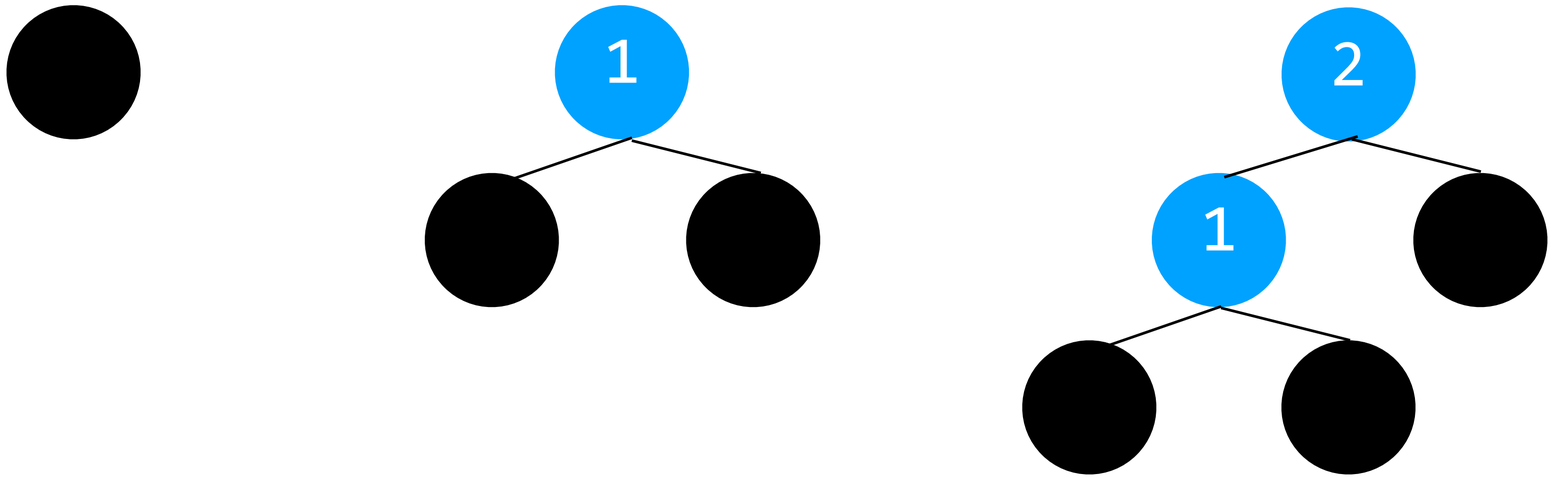
Lists

- A list is
  - the empty list, or
  - a pair of an element and a list.

- `Nil`, `[1] = Cons(1, Nil)`, `[2, 1] = Cons(2, [1])`, …

Binary Trees

- A binary tree is
  - the empty tree, or
  - a triple of a root element and two child trees.



Arithmetic Expressions

- An arithmetic expression is
  - a number,
  - a sum of two arithmetic expressions,
  - a difference of two arithmetic expressions,
  - a product of two arithmetic expressions, or
  - a ratio of two arithmetic expressions.

```
1
```

```
1 + 2
```

```
(1 + 2) * 1
```

```
(1 + 2) * 1 - 3
```

Arithmetic Expressions

- An arithmetic expression is, in general,
  - a number,
  - a pair of a unary operator and an arithmetic expression, or
  - a triple of a binary operator and two arithmetic expressions.

```
1        1 + 2        -(1 + 2)        -(1 + 2) * 3
```

Lambda Calculus

- An expression of lambda calculus (LC) is
  - a variable,
  - a lambda abstraction, which is a pair of a variable and an expression, or
  - a function application, which is a pair of two expressions.

```
x        λx.x        λx.x λx.x        λx.λy.(x y)
```

Implementation

- In C,
  - we can use structs and unions.

```
struct ae {
    int type;  // 0 if number, 1 if unary operator, and 2 if binary operator
    union {
        int n;
        char *op;
        struct ae *e0, *e1;
    } u;
};
```

Implementation

- In functional languages like OCaml and Haskell,
  - we can directly define such datatypes.

```
type ae =
    | Num of int
    | UnOp of string * ae
    | BinOp of string * ae * ae
```

Implementation

- In object-oriented languages,
  - we can use classes.

```
class AE
class Num(val n: Int) extends AE
class UnOp(val op: String, val e: AE) extends AE
class BinOp(val op: String, val e0: AE, val e1: AE)
  extends AE
```

Playing with AE

```
class AE

class Num(val n: Int) extends AE

class UnOp(val op: String, val e: AE) extends AE

class BinOp(val op: String, val e0: AE, val e1: AE) extends AE


new Num(1): AE

new UnOp("-", new Num(2)): AE

new BinOp("+", new Num(1), new UnOp("-", new Num(2))): AE
```

Playing with AE

```
class AE

class Num(val n: Int) extends AE

class UnOp(val op: String, val e: AE) extends AE

class BinOp(val op: String, val e0: AE, val e1: AE) extends AE


new AE: AE
```

- What does "`new AE`" represent?

Traits

- We need to prevent the creation of an instance of AE itself.

```
trait AE
class Num(val n: Int) extends AE
class UnOp(val op: String, val e: AE) extends AE
class BinOp(val op: String, val e0: AE, val e1: AE) extends AE

new AE  // trait AE is abstract; cannot be instantiated
```

Interpreting AE

- Implement an interpreter for AE.
  - `-` (negation) is only available operator for `UnOp`.
  - `+` (addition) is only available operator for `BinOp`.
  - Returns a resulting integer if success; throws an exception otherwise.

```
trait AE
class Num(val n: Int) extends AE
class UnOp(val op: String, val e: AE) extends AE
class BinOp(val op: String, val e0: AE, val e1: AE) extends AE
```

Interpreting AE

```
def interpret(e: AE): Int = {
  if (...)  // e is Num
    e.n
  else if (...)  // e is UnOp and e.op is "-"
    -interpret(e.e)
  else if (...)  // e is BinOp and e.op is "+"
    interpret(e.e0) + interpret(e.e1)
  else
    throws new Exception
}
```

Interpreting AE

```
def interpret(e: AE): Int = {
  if (e.isInstanceOf[Num])
    e.asInstanceOf[Num].n
  else if (e.isInstanceOf[UnOp] && e.asInstanceOf[UnOp].op == "-")
    -interpret(e.asInstanceOf[UnOp].e)
  else if (e.isInstanceOf[BinOp] && e.asInstanceOf[BinOp].op == "+")
    interpret(e.asInstanceOf[BinOp].e0) +
      interpret(e.asInstanceOf[BinOp].e1)
  else
    throws new Exception
}
```

Interpreting AE

- Dynamic type checking and casting via `isInstanceOf` and `asInstanceOf`
    - harm readability of code,
    - are verbose, and
    - are error-prone.
- Is there any better solution?

Method Overloading

- Can define more than one methods
    - of the same name
    - with different parameter types.
- A method is dispatched, i.e. selected, based on the types of arguments.

```
def println(n: Int): Unit = ...

def println(s: String): Unit = ...

println(1); println("1")
```

Interpreting AE

- Can method overloading be a solution?

```
def interpret(e: Num): Int = e.n

interpret(new Num(2))
```

Interpreting AE

- Can method overloading be a solution?

```
def interpret(e: UnOp): Int =
  if (e.op == "-") -interpret(e.e) else throw new Exception

def interpret(e: BinOp): Int =
  if (e.op == "+") interpret(e.e0) + interpret(e.e1)
  else throw new Exception
```

Interpreting AE

- Can method overloading be a solution?

```
def interpret(e: UnOp): Int =
  if (e.op == "-") -interpret(e.e) else throw new Exception
```

```
error: type mismatch;  found: AE;  required: UnOp
```

Interpreting AE

- Can method overloading be a solution?

```
def interpret(e: BinOp): Int =
  if (e.op == "+") interpret(e.e0) + interpret(e.e1)
  else throw new Exception
```

```
error: type mismatch;  found: AE;  required: BinOp
```

Method Overloading

- Method overloading is not a solution!
- A method is dispatched based on the "compile-time" types of arguments.
- Multiple dispatch is a semantics, which check the "run-time" types of arguments.
    - Most object-oriented languages do not use this semantics.

Visitor Pattern

- Actually, there is a solution in object-oriented style.
- You can use a visitor pattern.
- `en.wikipedia.org/wiki/Visitor_pattern`

- It does work but is verbose and requires a lot of boilerplate code.
- Today, we will try a more beautiful solution.

Pattern Matching

- In functional languages like OCaml and Haskell,
  - we can use pattern matching.

```
let rec interpret e =
  match e with
    | Num n -> n
    | UnOp ("-", e0) -> -(interpret e0)
    | BinOp ("+", e0, e1) -> (interpret e0) + (interpret e1)
    | _ -> raise Exception
```

Pattern Matching

- In Scala,
  - we can use pattern matching for types defined by classes.

```
def interpret(e: AE): Int = e match {
  case Num(n) => n
  case UnOp(op, e0) =>
    if (op == "-") -interpret(e0) else throw new Exception
  case BinOp(op, e0, e1) =>
    if (op == "+") interpret(e0) + interpret(e1)
    else throw new Exception
}
```

Pattern Matching

- In Scala,
  - we can use pattern matching for types defined by classes.

```
error: not found: value UnOp
error: not found: value UnOp
error: not found: value BinOp
```

- To use pattern matching, use case classes.

PLRG

Case Classes

```
trait AE

case class Num(n: Int) extends AE

case class UnOp(op: String, e: AE) extends AE

case class BinOp(op: String, e0: AE, e1: AE) extends AE
```

- Allow pattern matching.
- Do not require "`new`" when one create a new instance.
- Class parameters become fields automatically.
- The `toString`, `equal`, and `hashCode` methods are given.

Pattern Matching

```
def interpret(e: AE): Int = e match {
  case Num(n) => n
  case UnOp(op, e0) =>
    if (op == "-") -interpret(e0) else throw new Exception
  case BinOp(op, e0, e1) =>
    if (op == "+") interpret(e0) + interpret(e1)
    else throw new Exception
}
```

Pattern Matching

- Why pattern matching?
    - Is concise—no explicit type checking and casting.

Match Errors

```
def interpret(e: AE): Int = e match {
  case UnOp(op, e0) =>
    if (op == "-") -interpret(e0) else throw new Exception
  case BinOp(op, e0, e1) =>
    if (op == "+") interpret(e0) + interpret(e1)
    else throw new Exception
}
interpret(Num(3))  // MatchError: Num(3) (of class Num)
```

Checking Exhaustivity

```
sealed trait AE

case class Num(n: Int) extends AE

case class UnOp(op: String, e: AE) extends AE

case class BinOp(op: String, e0: AE, e1: AE) extends AE
```

- The "`sealed`" keyword makes classes or traits not extendable in different files.
- The compiler can check the exhaustivity of pattern matching.

Checking Exhaustivity

```
def interpret(e: AE): Int = e match {
  case UnOp(op, e0) =>
    if (op == "-") -interpret(e0) else throw new Exception
  case BinOp(op, e0, e1) =>
    if (op == "+") interpret(e0) + interpret(e1)
    else throw new Exception
}
```

```
warning: match may not be exhaustive.
It would fail on the following input: Num(_)
```

Pattern Matching

- Why pattern matching?
    - Is concise—no explicit type checking and casting.
    - Checks exhaustivity of patterns.

Wildcard and Contant Patterns

```scala
def grade(score: Int): String = {
  (score / 10) match {
    case 10 => "A"
    case 9 => "A"
    case 8 => "B"
    case 7 => "C"
    case 6 => "D"
    case _ => "F"
  }
}
```

```java
String grade(int score) {
  switch (score / 10) {
    case 10: return "A";
    case 9: return "A";
    case 8: return "B";
    case 7: return "C";
    case 6: return "D";
    default: return "F";
  }
}
```

Or Patterns

```scala
def grade(score: Int): String = {
  (score / 10) match {
    case 10 | 9 => "A"
    case 8 => "B"
    case 7 => "C"
    case 6 => "D"
    case _ => "F"
  }
}
```

```java
String grade(int score) {
  switch (score / 10) {
    case 10:
    case 9: return "A";
    case 8: return "B";
    case 7: return "C";
    case 6: return "D";
    default: return "F";
  }
}
```

Nested Patterns

```
def interpret(e: AE): Int = e match {
  case Num(n) => n
  case UnOp("-", e0) => -interpret(e0)
  case BinOp("+", e0, e1) => interpret(e0) + interpret(e1)
  case _ => throw new Exception
}
```

Optimizer for AE

```
def optimizeAdd(e: AE): AE = e match {
  case Num(_) => e
  case UnOp(op, e0) => UnOp(op, optimizeAdd(e0))
  case BinOp("+", Num(0), e1) => optimizeAdd(e1)
  case BinOp("+", e0, Num(0)) => optimizeAdd(e0)
  case BinOp(op, e0, e1) =>
    BinOp(optimizeAdd(e0), optimizeAdd(e1))
}
```

Optimizer for AE

```
def optimizeNeg(e: AE): AE = e match {
  case Num(_) => e
  case UnOp("-", UnOp("-", e0)) => e0
  case UnOp(op, e0) => UnOp(op, optimizeNeg(e0))
  case BinOp(op, e0, e1) =>
    BinOp(op, optimizeNeg(e0), optimizeNeg(e1))
}
```

Optimizer for AE

```
def optimizeAbs(e: AE): AE = e match {
  case Num(_) => e
  case UnOp("abs", e0 @ UnOp("abs", _)) => optimizeAbs(e0)
  case UnOp(op, e0) => UnOp(op, optimizeAbs(e0))
  case BinOp(op, e0, e1) =>
    BinOp(op, optimizeAbs(e0), optimizeAbs(e1))
}
```

Pattern Matching

- Why pattern matching?
    - Is concise—no explicit type checking and casting.
    - Checks exhaustivity of patterns.
    - Handles complex cases—wildcard and nested patterns.

Unreachable Patterns

```
def optimizeNeg(e: AE): AE = e match {
  case Num(_) => e
  case UnOp(op, e0) => UnOp(op, optimizeNeg(e0))
  case UnOp("-", UnOp("-", e0)) => e0
  case BinOp(op, e0, e1) =>
    BinOp(op, optimizeNeg(e0), optimizeNeg(e1))
}
```

```
warning: unreachable code
```

Pattern Matching

- Why pattern matching?
    - Is concise—no explicit type checking and casting.
    - Checks exhaustivity of patterns.
    - Handles complex cases—wildcard and nested patterns.
    - Checks the existence of unreachable patterns.

Type Patterns

```
def optimizeNeg(e: AE): AE = e match {
  case _: Num => e
  case UnOp("-", UnOp("-", e0)) => e0
  case UnOp(op, e0) => UnOp(op, optimizeNeg(e0))
  case BinOp(op, e0, e1) =>
    BinOp(op, optimizeNeg(e0), optimizeNeg(e1))
}
```

Type Patterns

```
def typeOf(a: Any): String = {
  a match {
    case n: Int => n + " is Int"
    case s: String => s + " is String"
    case _ => "I don't know! T.T"
  }
}
```

Tuple Patterns

```
sealed trait IList
case object INil extends IList
case class ICons(head: Int, tail: IList) extends IList

def equal_list(l0: IList, l1: IList): Boolean = {
  (l0, l1) match {
    case (ICons(h0, t0), ICons(h1, t1)) => h0 == h1 && equal_list(t0, t1)
    case (INil, INil) => true
    case _ => false
  }
}
```

Variable Declarations

```
val (n, m) = (1, 2)

val (a, b, c) = ("a", "b", "c")


val hd :: tl = List(1, 2, 3, 4)


val BinOp(op, e0, e1) = BinOp("+", Num(1), Num(2))
```

Anonymous Functions

```
val names = List("Jaemin", "Sukyoung", "Joonyoung", "Jihyeok")


names.zipWithIndex.foreach(
  p => println((p._2 + 1) + ". " + p._1)
)


names.zipWithIndex.foreach(p => p match {
  case (name, ind) => println((ind + 1) + ". " + name)
})
```

Anonymous Functions

```
val names = List("Jaemin", "Sukyoung", "Joonyoung", "Jihyeok")

names.zipWithIndex.foreach(p => p match {
  case (name, ind) => println((ind + 1) + ". " + name)
})

names.zipWithIndex.foreach {
  case (name, ind) => println((ind + 1) + ". " + name)
}
```

for Loops

```
names.zipWithIndex.foreach(
  p => println((p._2 + 1) + ". " + p._1)
)

for (p <- names.zipWithIndex)
  println((p._2 + 1) + ". " + p._1)

for ((name, ind) <- names.zipWithIndex)
  println((ind + 1) + ". " + name)
```

Command Line Arguments

```
def main(args: Array[String]): Unit =
  if (args.length == 2) {
      val from = args(0)
      val buf = read(from)
      val to = args(1)
      write(buf, to)
  } else
      println("usage: scala Copier.scala [from] [to]")
```

Sequence Patterns

```
def main(args: Array[String]): Unit =
  args match {
    case Array(from, to) =>
      val buf = read(from)
      write(buf, to)
    case _ =>
      println("usage: scala Copier.scala [from] [to]")
  }
```

Binary Search Trees

```
sealed trait BinTree

case object Empty extends BinTree

case class Node(n: Int, t0: BinTree, t1: BinTree) extends BinTree
```

Binary Search Trees

```
def add(t: BinTree, n: Int): BinTree =
  t match {
    case Empty => Node(n, Empty, Empty)
    case Node(m, t0, t1) =>
      if (n < m) Node(m, add(t0, n), t1)
      else if (n > m) Node(m, t0, add(t1, n))
      else t
  }
```

Pattern Guards

```
def add(t: BinTree, n: Int): BinTree =
  t match {
    case Empty => Node(n, Empty, Empty)
    case Node(m, t0, t1) if n < m => Node(m, add(t0, n), t1)
    case Node(m, t0, t1) if n > m => Node(m, t0, add(t1, n))
    case _ => t
  }
```

Binary Search Trees

```
def remove(t: BinTree, n: Int): BinTree =
  t match {
    case Empty => Empty
    ...
```

Binary Search Trees

```
def remove(t: BinTree, n: Int): BinTree =
  t match {
    ...
    case Node(m, t0, Empty) if n == m =>
      t0
    case Node(m, t0, t1: Node) if n == m =>
      val (min, t2) = removeMin(t1)
      Node(min, t0, t2)
    ...
```

Using Back Ticks

```
def remove(t: BinTree, n: Int): BinTree =
  t match {
    ...
    case Node(`n`, t0, Empty) =>
      t0
    case Node(`n`, t0, t1: Node) =>
      val (min, t2) = removeMin(t1)
      Node(min, t0, t2)
    ...
```

Binary Search Trees

```
def remove(t: BinTree, n: Int): BinTree =
  t match {
    ...
    case Node(m, t0, t1) if n < m =>
      Node(m, remove(t0, n), t1)
    case Node(m, t0, t1) if n > m =>
      Node(m, t0, remove(t1, n))
  }
```

PLRG

Summary

- In many cases, a single datatype contains heterogenous shapes of values.
- One can implement such types via sealed trait and case classes/objects.
- Pattern matching treats values of such types effectively.
    - Is concise and safe.
    - Resolves complex cases through nested patterns.
    - Checks the exhaustivity and reachability of patterns.