Linguagens de Programação

Fabio Mascarenhas - 2017.2

http://www.dcc.ufrj.br/~fabiom/lp

Tipos algébricos

- Recapitulando, uma lista é definida como a lista vazia ou um par composto de um elemento e outra lista

- Esse tipo de definição de uma estrutura de dados é uma instância de um padrão mais geral: os *tipos algébricos*

- A definição de um tipo algébrico é dada por um ou mais *construtores*, onde os construtores que têm parâmetros do próprio tipo algébrico são os *casos indutivos*, e os construtores que não têm parâmetros do próprio tipo algébrico são os *casos base*

- Para listas, o construtor `Nil` é o único caso base, e o construtor `::` é o único caso indutivo

Case classes

- Tipos algébricos se prestam naturalmente à desconstrução por casamento de padrões, e por isso linguagens funcionais costumam oferecer sintaxe para sua definição

- Em Scala, definimos tipos algébricos usando *case classes*

```
trait Lista[T]
case class Vazia[T]() extends Lista[T]
case class Cons[T](hd: T, tl: Lista[T]) extends Lista[T]

def tamanho[T](l: Lista[T]): Int = l match {
  case Vazia() => 0
  case Cons(hd, tl) => 1 + tamanho(tl)
}
```

Case classes

- A declaração do tipo algébrico é dada por um *trait*
- Cada construtor é uma *case class* que extende o tipo algébrico, e lista seus parâmetros
- O construtor define tanto uma função para construir elementos do tipo algébrico, quanto um padrão para ser usado no *match*
- Podemos definir as funções que operam em um tipo algébrico dentro do seu *trait* também, e usá-las com a sintaxe OO de Scala

Case classes estilo OO

- Definindo uma função *map* em Lista[T] com a sintaxe OO:

```
trait Lista[T] {
  def map[U](f: T => U): Lista[U] = this match {
    case Vazia() => Vazia[U]
    case Cons(hd, tl) => Cons(f(hd), tl.map(f))
  }
}
case class Vazia[T]() extends Lista[T]
case class Cons[T](hd: T, tl: Lista[T]) extends Lista[T]
```

`Option[T]`

- O tipo `Option[T]` é outro tipo algébrico pré-definido em Scala, para representar valores opcionais de maneira mais segura que o uso de *null*

- Um `Option[T]` pode ser ou `None`, que quer dizer que não há nenhum valor, ou `Some(x: T)`

- Podemos usar `Option[T]` para criar uma versão segura da função que obtém o primeiro elemento de uma lista:

```
def primeiro[T](l: List[T]): Option[T] = l match {
  case Nil => None
  case h :: t => Some(h)
}
```

Options e `for`

- Uma maneira de enxergar um `Option[T]` é como uma lista contendo no máximo um elemento do tipo `T`

- Em Scala, o tipo `Option[T]` também implementa algumas funções que vimos para listas, em especial `flatMap` e `filter`

- Isso quer dizer que podemos usar a sintaxe do `for` com geradores que retornam `Option[T]` também:

```
def multPrimeiro(l1: List[Int], l2: List[Int]): Option[Int] =
  for {
    x <- primeiro(l1)
    y <- primeiro(l2)
  } yield x * y
```

Compilando `for`

- Uma expressão `for` é apenas açúcar sintático!

```
for {
  p <- e1
  if e2
  ...
} yield e3
```



```
for {
  p <- e1.filter({ case p => e2 })
  ...
} yield e3
```

```
for {
  p <- e1
  ...
} yield e2
```



```
e1.flatMap({ case p => for {
  ...
} yield e2 })
```

```
for {
  p <- e1
} yield e2
```



```
 e1.map({ case p => e2 })
```

Árvores

- Uma aplicação comum para tipos algébricos são estruturas em árvore

- Por exemplo, uma árvore binária rotulada é simples de definir:

```
trait ArvoreBin[T]
case class Folha[T](rot: T) extends ArvoreBin[T]
case class Ramo[T](rot: T,
                   esq: ArvoreBin[T],
                   dir: ArvoreBin[T]) extends ArvoreBin[T]
```

- Claro que variantes são possíveis, como árvores sem rótulos nas folhas, e árvores com rótulos apenas nas folhas

map e fold em árvores

- É bem fácil definir um equivalente de `map` para nossas árvores binárias:

```
trait ArvoreBin[T] {
  def map[U](f: T => U): ArvoreBin[U] = this match {
    case Folha(x) => Folha(f(x))
    case Ramo(x, e, d) => Ramo(f(x), e.map(f), d.map(f))
  }
}
```

- Um *fold* é mais complicado; uma maneira seria definir folds equivalentes aos folds resultantes da lista que temos quando caminhamos a árvore em pré-ordem, ordem ou pós-ordem

Catamorfismos

- Catamorfismos são generalizações da operação *fold* para outros tipos algébricos
- Um catamorfismo é a substituição dos construtores de um tipo algébrico por outras funções
- No caso do fold de listas, o construtor `Nil` é substituído por uma [função] constante `z` e o construtor `::` por uma função binária `f`
- Logo, para árvores binárias, vamos substituir o construtor `Folha` por uma função de um parâmetro `f` e o construtor `Ramo` por uma função de três parâmetros `r`

Catamorfismos vs Casamento de Padrões

- Um fold ou catamorfismo é parecido com um casamento de padrões simples para uma estrutural de dados algébrica
- Temos apenas um caso para cada construtor, correspondendo a cada função passada no catamorfismo
- A diferença é na recursão: no casamento de padrões fazemos a recursão explicitamente, enquanto no catamorfismo ela está implícita

fun - uma mini-linguagem funcional

- Agora que vimos como se *usa* uma linguagem funcional como Scala, vamos estudar como se dá a *semântica* de uma linguagem funcional
- Vamos transformar nosso modelo informal de execução em um modelo *preciso*
- Para isso, vamos construir aos poucos um *interpretador* para uma linguagem funcional simples
- Um interpretador é uma função que vai levar um programa fun (uma árvore representando as expressões do programa) em um valor

fun - Aritmética

- Sintaxe concreta vs abstrata

```
exp : NUM
    | exp ‘+’ exp
    | exp ‘*’ exp
    | ‘(’ exp ‘)’
```



```
trait Exp
case class Num(v: Double) extends Exp
case class Soma(e1: Exp, e2: Exp) extends Exp
case class Mult(e1: Exp, e2: Exp) extends Exp
```

- Um *parser* converte, por ex, “2+2*3” em `Soma(Num(2), Mult(Num(2), Num(3)))`

fun - Aritmética

- O interpretador de fun pode ser facilmente definido com uma função `eval` dentro de `Exp`, usando casamento de padrões
- O que são números em fun? Números de ponto flutuante de precisão dupla. Por quê? Porque podemos simplesmente usar `Doubles` em Scala e a aritmética de Scala para interpretar fun
- Outras representações para números (por ex., inteiros com precisão arbitrária) levariam a outros interpretadores
- A linguagem em que estamos definindo o interpretador influencia a linguagem interpretada, a não ser que tomemos bastante cuidado!