Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

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 ')'
```

E → T { + T }
T → F { * F }
F → num | (E)

```
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)))`

Disgressão – Parser de Combinadores

- Não é difícil expressar o parser diretamente em uma linguagem funcional, através de *combinadores*

- Um combinador é apenas outro nome para uma função de alta ordem que recebe uma ou mais funções e retorna uma outra função, todas com o mesmo "formato"

- Podemos enxergar um parser como uma função da entrada para a saída do parser, e os combinadores são as diferentes formas de compor um parser (sequência, escolha, opcional, repetição, etc.)

- Nossos parsers vão ser um pouco mais ricos, para termos boa informação em caso de erro de sintaxe

O tipo `Parser[A]`

- Podemos modelar um parser como uma função, mas para ter acesso à expressão for temos uma *classe ímplicita* associada

input pos fail expected

```
type Parser[A] = (Vector[Char], Int, Int, Set[String]) =>
          (Option[(A, Int)], Int, Set[String])
```

alias de tipo

output pos fail expected

```
implicit class RichParser[A](val p: Parser[A]) extends AnyVal {
   def flatMap[B](f: A => Parser[B]): Parser[B] = bind(p, f)
   def map[B](f: A => B): Parser[B] = adapt(p, f)
   def filter(f: A => Boolean): Parser[A] = constrain(p, f)
}
```

classe implícita associada

```
def empty[A](v: A): Parser[A] = ???
def pred(pred: Char => Boolean): Parser[Char] = ???
def choice[A](p1: Parser[A], p2: Parser[A]): Parser[A] = ???
def bind[A,B](p: Parser[A], f: A => Parser[B]): Parser[B] = ???
def adapt[A,B](p: Parser[A], f: A => B): Parser[B] = ???
def constrain[A](p: Parser[A], f: A => Boolean): Parser[A] = ???
def not[A,B](p: Parser[A], v: B): Parser[B] = ???
def expect[A](p: Parser[A], name: String): Parser[A] = ???
val pos: Parser[Int] = ???
```

primitivos

Combinadores primitivos e derivados

- As funções do slide anterior são as primitivas de parsing
- Usando elas podemos definir vários outros combinadores úteis
- Por exemplo: reconhecer um caractere, transformar uma `List[Parser[A]]` em um `Parser[List[A]]`, reconhecer zero ou mais ocorrências de um `Parser[A]`, reconhecer uma ou nenhuma ocorrência de um `Parser[A]`, fazer um fold à esquerda de uma sequência de `Parser[A]` intercalados por um operador `Parser[(A,A) => A]`
- Podemos também construir parsers para os tokens de *fun*: espaço em branco, palavras-chave, identificadores, operadores e numerais

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!

Big-step vs small-step

- A função `eval` é uma definição *big-step* do significado de uma expressão fun

- Ela ainda deixa algumas coisas nebulosas: por exemplo, em uma soma qual expressão é avaliada primeiro?

- Há maneiras de tornar uma definição big-step mais precisa que vamos ver depois, mas também podemos dar o significado de uma expressão com uma definição *small-step*

- Uma definição small-step é um *passo de avaliação*, levando de uma expressão fun para uma outra expressão fun

- Quando chegamos em uma expressão que avalia para ela própria terminamos

fun small-step

- Vamos definir uma função `step` que dá um passo de avaliação
- Uma expressão que avalia em um passo para ela mesma é chamada de *forma normal*
- Por enquanto, as única formas normais de fun são expressões `Num`
- Formas normais de uma definição small-step correspondem aos valores resultado de uma definição big-step

Açúcar Sintático

- Podemos acrescentar expressões de subtração e negação a *fun* com modificações simples no parser e na sintaxe abstrata

- Mudar o interpretador (acrescentando casos novos) também não seria difícil, mas vamos implementar esses novos termos via *açúcar sintático*

- A transformação é bem simples: `e1 - e2 => e1 + -1 * e2` e `-e => -1 * e`

- Em nossa linguagem, tanto subtração quanto negação são *açúcar sintático*: uma transformação puramente local de expressões em uma linguagem extendida para uma linguagem mais simples

- Em geral açúcar sintático é implementado direto no parser!

Condicionais

- Para ter mais poder em nossa linguagem, vamos agora introduzir um operador relacional `<` e uma expressão condicional `if`

```
exp : ...
    | exp ‘<’ exp
    | IF exp THEN exp ELSE exp END
```

```
case class Menor(e1: Exp, e2: Exp) extends Exp
case class If(cond: Exp, ethen: Exp, eelse: Exp) extends Exp
```

- Temos um problema: qual deve ser o resultado de `<`? Como o `if` avalia para uma expressão ou para outra?

Booleanos

- Poderíamos adotar a estratégia de C, e dizer que `e1 < e2` é 1 se o valor de `e1` for menor que `e2`, e 0 se não for

- Mas vamos introduzir um novo tipo de dado em *fun*: booleanos

- O interpretador agora não pode mais produzir um `Double`, precisamos de um tipo algébrico para os valores de *fun* (e formas normais correspondentes para a definição small-step)

```
trait Valor
case class NumV(v: Double) extends Valor
case class TrueV() extends Valor
case class FalseV() extends Valor
```

e os erros?

Erros

- Algumas operações de fun só são válidas para determinados operandos: soma, multiplicação e menor só são válidas para números, e um condicional só é válido se a condição for booleana

```
case Soma(e1, e2) => {
  val (NumV(v1), NumV(v2)) = (eval(e1), eval(e2))
  NumV(v1 + v2)
}
```

- Uma definição como a acima, que assume que os operandos estão corretos, diz que operações com operandos inválidos são *indefinidas*

- Uma operação indefinida não tem resultado

- Em geral, linguagens com operações indefinidas contam com um *verificador de tipos* que rejeita programas que poderiam ter operações indefinidas em tempo de execução

Avaliação checada

- Uma alternativa a operações indefinidas é levar erros em conta na própria definição, adicionando um *valor de erro* (e sua forma normal correspondente em uma definição small-step)

```
case Soma(e1, e2) => (evalc(e1), evalc(e2)) match {
  case (NumV(v1), NumV(v2)) => NumV(v1 + v2)
  case _ => ErroV()
}
```

- Na definição small-step:

```
case Soma(_, True()) => Erro()
case Soma(_, False()) => Erro()
case Soma(True(), _) => Erro()
case Soma(False(), _) => Erro()
```

- Sem checagem como acima, uma operação indefinida leva a avaliação small-step a ficar *stuck* (travada): uma expressão avalia em um passo para ela mesma sem ser uma forma normal