Linguagens de Programação

Fabio Mascarenhas - 2017.2

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

fun - Aritmética

- Sintaxe concreta vs abstrata

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

```
trait Exp
case class Num(v: String) 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!

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 um `Num`

fun small-step

- Vamos definir uma função `step` que dá um passo de avaliação
- Uma expressão que é um valor da linguagem é uma *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 incluir as formas normais nesse tipo)

```
trait Valor
case class Num(v: Double) extends Exp, Valor
case class Bool(v: Boolean) extends Exp, Valor
```

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 (Num(v1), Num(v2)) = (eval(e1), eval(e2))
  Num(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