Implementação de Linguagens

(Módulo sobre Linguagens Funcionais)

Pedro Vasconcelos

DCC/FCUP

3 de Abril de 2020

Objectivos

- Introdução à implementação de linguagens de programação funcionais.
- Enfoce sobre técnicas de interpretação/compilação usando máquinas virtuais.

Pré-requisitos

- Conhecimento da linguagem Haskell (ou ML/Caml/F#).
- Conhecimento de linguagem C.
- Noções de técnicas de compilação:
  - análise léxica;
  - análise sintática (*parsing*);
  - árvores sintáticas abstractas;
  - geração de código.

Funcionamento

- Aulas teóricas-práticas: 2 $\times$ 1.5 h por semana (video-conferência)
- Dois módulos:

  Linguagens lógicas (Ricardo Rocha – até 31 março)
  Linguagens funcionais (Pedro Vasconcelos — a partir de 3 abril)

Avaliação

Trabalho prático:

- Cotado para 4 valores (8 valores para os 2 trabalhos)

Exame final:

- Cotado para 12 valores em 20.
- Classificação mínima de 40% no exame

Bibliografia principal

- *Foundations for Functional Programming*, Lawrence C. Paulson, `http://www.cl.cam.ac.uk/~lp15/`. Introdução ao cálculo-λ (próxima aula).
- *The Implementation of Functional Languages*, Simon Peyton Jones, `https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-lan`

Revisões de Haskell:

- *Programming in Haskell*, Graham Hutton, Cambridge University Press, 2007.
- *Learn you a Haskell for great good*, Miran Lipovaca, `http://learnyouahaskell.com/`

Programação funcional

- No paradigma imperativo a computação é expressa pela execução de instruções que mudam variáveis (estado)
- No paradigma funcional a computação é expressa usando aplicação de funções a valores
- As linguagens funcionais suportam e promovem o estilo funcional
- Exemplos: Scheme, ML, O'Caml, F#, Haskell, Scala

Linguagens funcionais modernas

- Funções são *valores de primeira classe*:
  - podem ser passadas a outras funções (ordem superior)
  - podem ser guardadas em estruturas de dados
- Estruturas de dados:
  - enumerações, tuplos, listas, árvores...
  - decomposição usando encaixe de padrões
  - gestão de memória automática
- Sistema de tipos estáticos
  - tipos determinados antes da execução
  - inferência automática de tipos (total ou parcial)
- Suporte para programação em larga escala
  - módulos (ML, O'Caml, Haskell)
  - classes de tipos (Haskell)
  - objectos (O'Caml, F#, Scala)

Estado da arte

- Nos últimos 30 anos: muita investigação académica
- Implementações robustas (e.g., O'Caml, Glasgow Haskell Compiler, F#, Scala)
- Bastante eficientes:[1]
  - 50%-90% velocidade de C em *bechmarks* sintéticos
  - Performance semelhante ao Java com JIT
  - Muito boa performance em alguns casos (e.g. concorrência)
- Nos últimos 10-15 anos: crescente utilização industrial
  - Jane Street, Standard Chartered, Facebook, Microsoft, Github...

[1] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/ghc-clang.html`

Metodologia para estas aulas

1. Formalização de linguagens: sintaxe e semântica
2. Fundamentos de programação funcional (*cálculo-*λ)
3. FUN: uma linguagem funcional mínima
   - ilustrar princípios de desenho e implementação
   - permitir a experimentação por parte dos alunos
4. Semânticas formais para FUN
5. compiladores/interpretadores em Haskell/C:
   - programas sucintos
   - próximos do formalismo matemático
   - especificação executável
   - base para trabalhos práticos

Formalização de linguagens

Sintaxe	regras de formação de expressões, frases, etc.
Semântica	definição do significado desses fragmentos

A sintaxe é formalizada usando gramáticas livres de contexto.

A semântica pode ser formalizada de vária formas; vamos usar semânticas operacionais.

Gramáticas livres de contexto

Σ conjunto de símbolos terminais.

V conjunto de símbolos não-terminais.

P conjunto de produções da forma

$$\text{<não-terminal>} ::= \text{<alt>}_1 \quad | \quad \ldots \quad | \quad \text{<alt>}_n$$

em que <alt>_i são sequências de terminais ou não-terminais.

$S \in V$ símbolo não-terminal inicial

Linguagem sequências de símbolos terminais geradas apartir de S usando as produções.

Exemplo: expressões aritméticas

Terminais $\Sigma = \{0, 1, \ldots, 9, +, *, (,)\}$

Não-terminais $V = \{E, N, A\}$

Símbolo inicial E

Produções

$$
\begin{aligned}
E &::= E + E \mid E * E \mid (E) \mid N \\
N &::= AN \mid A \\
A &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid \\
&\quad\ 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

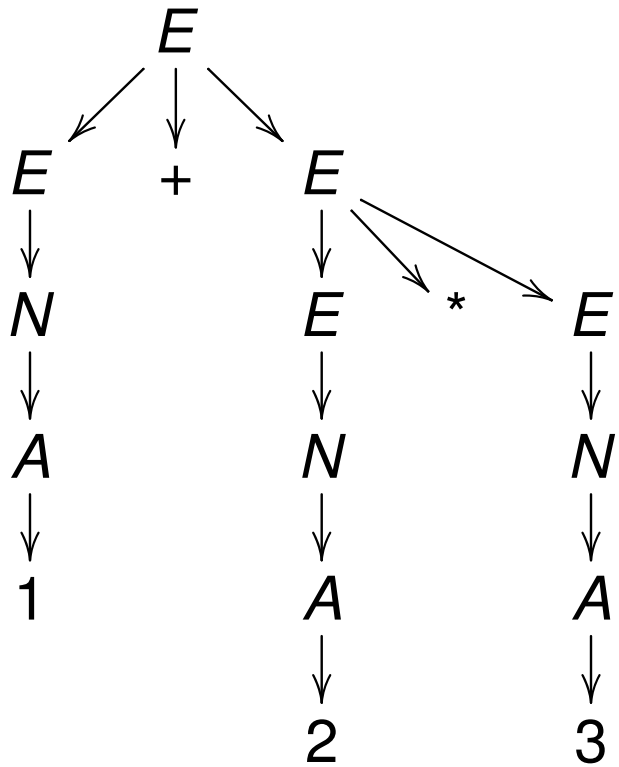
Derivações

$$
\begin{aligned}
E &\rightarrow E + E \rightarrow E + E * E \rightarrow N + E * E \rightarrow A + E * E \\
&\rightarrow 1 + E * E \rightarrow 1 + N * E \rightarrow 1 + A * E \rightarrow 1 + 2 * E \\
&\rightarrow 1 + 2 * N \rightarrow 1 + 2 * A \rightarrow \underbrace{1 + 2 * 3}_{\text{terminais}}
\end{aligned}
$$

Logo: “1+2*3” é uma palavra da linguagem de expressões.

Árvores de derivação

Podemos representar a derivação de uma palavra por uma árvore:



Exercício: encontrar *outra* derivação que resulta numa *árvore diferente*.[2]

[2] Esta gramática é *ambígua*.

Sintaxe abstracta

A sintaxe abstracta representa a estrutura da árvore de derivação omitindo informação desnecessária, e.g.:

- decomposição de números em algarismos
- parêntesis

Definindo um tipo de dados em Haskell:

```
data Expr = Add Expr Expr
          | Mult Expr Expr
          | Num Int
          deriving (Eq, Show)
```

Sintaxe concreta vs. abstracta

```
1+2*3
Add (Num 1) (Mult (Num 2) (Num 3))
Mult (Add (Num 1) (Num 2)) (Num 3)

(1+2)*3
Mult (Add (Num 1) (Num 2)) (Num 3)

1+2+3
Add (Add (Num 1) (Num 2)) (Num 3)
Add (Num 1) (Add (Num 2) (Num 3))
```

Semântica operacional

- Uma relação $\Rightarrow$ entre expressões e valores (inteiros)
- Definida indutivamente sobre termos da sintaxe abstracta
- Regras de inferência da forma

$$\frac{A_1 \quad A_2 \quad \ldots \quad A_n}{B}$$

$A_1, A_2, \ldots, A_n$ hipóteses
B conclusão
axioma quando $n = 0$
leitura lógica: se $A_1, \ldots, A_n$ então B
leitura computacional: para obter B, efectuar $A_1, \ldots, A_n$.

Regras de inferência

$$\frac{}{\text{Num } n \Rightarrow n}$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\text{Add } e_1\ e_2 \Rightarrow n_1 + n_2}$$

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\text{Mult } e_1\ e_2 \Rightarrow n_1 \times n_2}$$

Exemplo duma derivação

$$\dfrac{\dfrac{\dfrac{}{\text{Num } 1 \Rightarrow 1} \qquad \dfrac{}{\text{Num } 2 \Rightarrow 2}}{\text{Add (Num 1) (Num 2)} \Rightarrow 3} \qquad \dfrac{}{\text{Num } 3 \Rightarrow 3}}{\text{Mul (Add (Num 1) (Num 2)) (Num 3)} \Rightarrow 9}$$

Interpretador em Haskell

Podemos implementar $\Rightarrow$ como uma função recursiva:

```
eval :: Expr -> Int
eval (Num n)   = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mult e1 e2) = eval e1 * eval e2
```

Exemplo:

```
  eval (Mult (Add (Num 1) (Num 2)) (Num 3))
= eval (Add (Num 1) (Num 2)) * eval (Num 3)
= (eval (Num 1) + eval (Num 2)) * 3
= (1 + 2) * 3
= 9
```

Correção

O interpretador implementa correctamente a semântica operacional:

$$e \Rightarrow n \text{ se e só se eval } e = n$$

Prova: por *indução estrutural* sobre e.

Compilação de expressões

- Vamos definir uma *máquina virtual* para a linguagem das expressões aritméticas
- Traduzimos cada expressão numa *sequência de operações*
- Eliminamos a recursão: os valores intermédios passam a ser explícitos
- Para executar o código virtual já não necessitamos da árvore sintática

Uma máquina de pilha

Configuração da máquina abstracta:

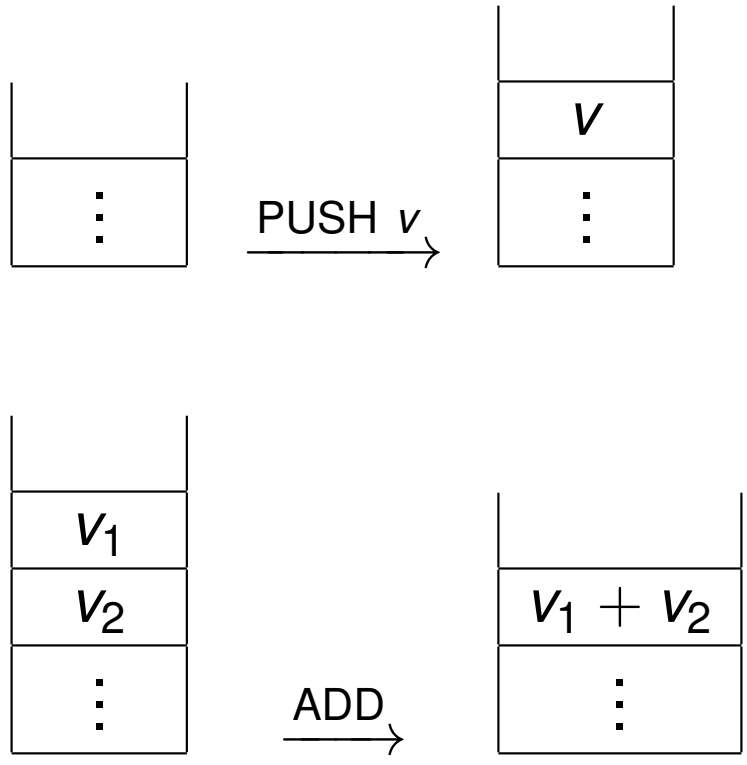
pilha: uma sequência de inteiros;

programa: uma sequência de *instruções*

Instruções da máquina abstracta:

PUSH *n*	coloca um valor no topo da pilha
ADD, MUL	operações aritméticas (sobre valores na pilha)

Operações sobre a pilha



Analogamente para a instrução MUL.

Definições da máquina

```
-- instruções da máquina virtual
data Instr = PUSH Int
           | ADD
           | MUL
           deriving (Eq,Show)

-- a pilha é uma lista de valores
type Stack = [Int]

-- o código é uma lista de instruções
type Code = [Instr]

-- a configuração da máquina
type State = (Stack, Code)
```

Compilador de expressões

```
compile :: Expr -> Code
compile (Num n) = [PUSH n]
compile (Add e1 e2)
     = compile e1 ++ compile e2 ++ [ADD]
compile (Mul e1 e2)
     = compile e1 ++ compile e2 ++ [MUL]
```

- Traduz uma expressão numa sequência de instruções
- Análogo a *eval*, mas gera instruções em vez valores

Invariante

A execução de `compile e` acrescenta o valor de `e` ao topo da pilha.

Exemplo

```
> compile (Mult (Add (Num 1) (Num 2)) (Num 3))
[PUSH 1, PUSH 2, ADD, PUSH 3, MUL]
```

Função de transição

Implementa a transição associada a cada instrução da máquina abstracta:

```
exec :: State -> State
exec (stack, PUSH v:code) = (v:stack, code)
exec (v1:v2:stack, ADD:code)
                        = ((v1+v2):stack, code)
exec (v1:v2:stack, MUL:code)
                        = ((v1*v2):stack, code)
```

Interpretador de código virtual

Iterar a função de transição até esgotar as instruções; o resultado é o topo da pilha.

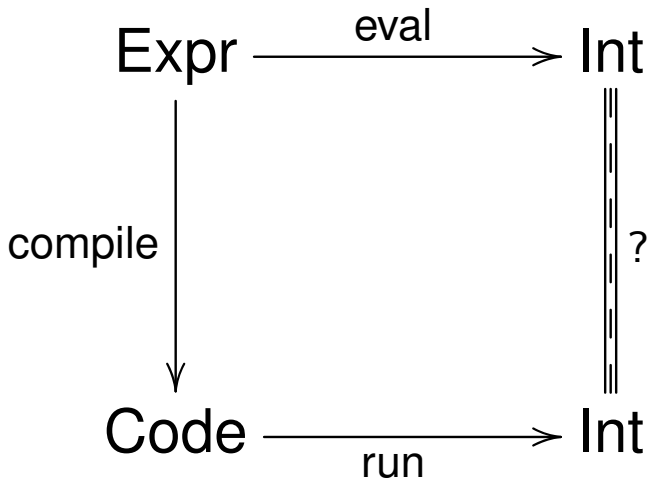
```
runState :: State -> Int
runState (v:_, []) = v
runState s         = runState (exec s)

run :: Code -> Int
run c = runState ([],c)
```

Exemplo de execução:

```
> run (compile (Mult(Add(Num 1)(Num 2))(Num 3))
9
```

Dois processos para calcular expressões



Correção da compilação

```
∀e. eval e = run (compile e)
```

Conclusão

- Dois interpretadores em Haskell:
  - **eval**
    - um interpretador de expressões
    - especificação de alto-nível
  - **run**
    - interpretador de código virtual
    - mais perto de uma máquina real
- Correção da compilação: os dois interpretadores produzem o mesmo resultado para qualquer expressão
- Na próxima aula, vamos (re)ver o cálculo-λ:
  - um modelo abstracto duma linguagem de programação
  - variáveis, funções, aplicação, recursão...