Uma Linguagem Funcional

(Reynolds, Theories of Programming Languages)

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2019/2020

Uma Linguagem Funcional

Apresentam-se agora duas linguagens funcionais que estendem o lambda calculus com operações lógicas e aritméticas, expressões condicionais, definições, tuplos, alternativas e listas.

- Uma linguagem estrita, com uma semântica de avaliação aplicativa ("call-by-value").
- Uma linguagem não estrita, com uma semântica de avaliação normal ("call-by-name").

Uma Linguagem Funcional "call-by-value"

Sintaxe abstracta

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle var \rangle \mid \lambda\langle var \rangle.\langle exp \rangle \mid \langle exp \rangle\langle exp \rangle \\
& \mid & 0 \mid 1 \mid 2 \mid \ldots \mid \textsf{True} \mid \textsf{False} \\
& \mid & \neg\langle exp \rangle \mid -\langle exp \rangle \mid \langle exp \rangle \ \mathbf{bop} \ \langle exp \rangle \\
& \mid & \textsf{if } \langle exp \rangle \textsf{ then } \langle exp \rangle \textsf{ else } \langle exp \rangle \\
& \mid & \langle\langle exp \rangle, \ldots, \langle exp \rangle\rangle \mid \langle exp \rangle.\langle tag \rangle \\
& \mid & @\langle tag \rangle\langle exp \rangle \mid \textsf{sumcase } \langle exp \rangle \textsf{ of } (\langle exp \rangle, \ldots, \langle exp \rangle) \\
& \mid & \textsf{nil} \mid \langle exp \rangle :: \langle exp \rangle \mid \textsf{listcase } \langle exp \rangle \textsf{ of } (\langle exp \rangle, \langle exp \rangle) \\
& \mid & \lambda\langle pat \rangle.\langle exp \rangle \mid \textsf{let } \langle pat \rangle \equiv \langle exp \rangle, \ldots, \langle pat \rangle \equiv \langle exp \rangle \textsf{ in } \langle exp \rangle \\
& \mid & \textsf{letrec } \langle var \rangle \equiv \lambda\langle pat \rangle.\langle exp \rangle, \ldots, \langle var \rangle \equiv \lambda\langle pat \rangle.\langle exp \rangle \textsf{ in } \langle exp \rangle
\end{array}
$$

$$\mathbf{bop} \in \{+, -, *, \textsf{div}, \textsf{mod}, =, \neq, <, >, \leq, \geq, \vee, \wedge\}$$

$$
\begin{array}{rcl}
\langle tag \rangle & ::= & 1 \mid 2 \mid \ldots \\
\langle pat \rangle & ::= & \langle var \rangle \mid \langle\langle pat \rangle, \ldots, \langle pat \rangle\rangle
\end{array}
$$

Precedências e associatividade dos operadores

- Para evitar o uso excessivo de parêntesis estipulamos a seguinte lista de precedências. A associatividade é à esquerda, excepto nos casos assinalados.
  - .
  - aplicação, @
  - *, div, mod
  - - , $+$
  - $=, \neq, <, >, \leq, \geq$
  - $\neg$
  - $\wedge$
  - $\vee$
  - $::$ (associativa à direita)

Formas canónicas

- A linguagem usa uma semântica de avaliação "call-by-value" (CBV).
- Serão dadas regras de inferência para a relação de avaliação $\Rightarrow_E$ (escreveremos abreviadamente $\Rightarrow$).
- $\Rightarrow$ relaciona as expressões com as formas canónicas da linguagem.
- Teremos formas canónicas de diferentes tipos

$$
\begin{array}{lcl}
\langle cfm \rangle & ::= & \lambda \langle var \rangle . \langle exp \rangle \\
& | & \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
& | & \mathsf{True} \mid \mathsf{False} \\
& | & \langle \langle cfm \rangle, \ldots, \langle cfm \rangle \rangle \\
& | & @\langle tag \rangle \langle cfm \rangle \\
& | & \mathsf{nil} \mid \langle cfm \rangle :: \langle cfm \rangle
\end{array}
$$

Notação

- Avaliação CBV: $\langle exp \rangle \Rightarrow \langle cfm \rangle$
- Se a avaliação de uma expressão e não terminar ou bloquear, escreveremos simplemente $e\uparrow$.
- Na apresentação das regras da semântica CBV usaremos as seguintes meta-variáveis

$$
\begin{array}{llcll}
e & \langle exp \rangle_{\mathsf{closed}} & \quad & i & \mathbb{Z} \\
\widehat{e} & \langle exp \rangle & & k, n & \mathbb{N} \\
z & \langle cfm \rangle & & b & \mathbb{B} \\
v, u & \langle var \rangle & & p & \langle pat \rangle
\end{array}
$$

- $\lfloor i \rfloor$ e $\lfloor b \rfloor$ denotam as formas canónicas das expressões i e b.
  - Ex: se i denota o inteiro 3 e i' o inteiro 2, $\lfloor i + i' \rfloor$ denota o inteiro 5.

Semântica de avaliação "call-by-value"

- **Formas canónicas**

$$\frac{}{z \Rightarrow z}$$

- **Aplicação**

$$\frac{e \Rightarrow \lambda v.\widehat{e} \qquad e' \Rightarrow z' \qquad \widehat{e}[z'/v] \Rightarrow z}{e\,e' \Rightarrow z}$$

Semântica de avaliação “call-by-value”

- **Operadores unários**

$$\frac{e \Rightarrow \lfloor i \rfloor}{-e \Rightarrow \lfloor -i \rfloor}$$

$$\frac{e \Rightarrow \lfloor b \rfloor}{\neg e \Rightarrow \lfloor \neg b \rfloor}$$

Semântica de avaliação “call-by-value”

- **Operadores binários**

$$\frac{e_1 \Rightarrow \lfloor i_1 \rfloor \qquad e_2 \Rightarrow \lfloor i_2 \rfloor}{e_1 \ \mathbf{bop}\ e_2 \Rightarrow \lfloor i_1 \ \mathbf{bop}\ i_2 \rfloor} \quad \text{para } \mathbf{bop} \in \{+, -, *, =, \neq, <, >, \leq, \geq\}$$

$$\frac{e_1 \Rightarrow \lfloor i_1 \rfloor \qquad e_2 \Rightarrow \lfloor i_2 \rfloor}{e_1 \ \mathbf{bop}\ e_2 \Rightarrow \lfloor i_1 \ \mathbf{bop}\ i_2 \rfloor} \quad \text{para } \mathbf{bop} \in \{\mathsf{div}, \mathsf{mod}\} \text{ e } i_2 \neq 0$$

$$\frac{e_1 \Rightarrow \lfloor b_1 \rfloor \qquad e_2 \Rightarrow \lfloor b_2 \rfloor}{e_1 \ \mathbf{bop}\ e_2 \Rightarrow \lfloor b_1 \ \mathbf{bop}\ b_2 \rfloor} \quad \text{para } \mathbf{bop} \in \{\vee, \wedge\}$$

Semântica de avaliação “call-by-value”

- **Expressões condicionais**

$$\frac{e_1 \Rightarrow \mathsf{True} \qquad e_2 \Rightarrow z}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow z}$$

$$\frac{e_1 \Rightarrow \mathsf{False} \qquad e_3 \Rightarrow z}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow z}$$

Semântica de avaliação “call-by-value”

- **Tuplos**

$$\frac{e_1 \Rightarrow z_1 \quad \cdots \quad e_n \Rightarrow z_n}{\langle e_1, \ldots, e_n \rangle \Rightarrow \langle z_1, \ldots, z_n \rangle}$$

$$\frac{e \Rightarrow \langle z_1, \ldots, z_n \rangle}{e.k \Rightarrow z_k} \quad \text{para } k \in \{1, \ldots, n\}$$

Semântica de avaliação "call-by-value"

- **Alternativas**

$$\frac{e \Rightarrow z}{@k\, e \Rightarrow @k\, z}$$

$$\frac{e \Rightarrow @k\, z \qquad e_k\, z \Rightarrow z'}{\text{sumcase } e \text{ of } (e_1, \ldots, e_n) \Rightarrow z'} \quad \text{para } k \in \{1, \ldots, n\}$$

Semântica de avaliação "call-by-value"

- **Listas**

$$\frac{}{\text{nil} \Rightarrow \text{nil}} \qquad \frac{e \Rightarrow z \qquad e' \Rightarrow z'}{e :: e' \Rightarrow z :: z'}$$

$$\frac{e \Rightarrow \text{nil} \qquad e_{\text{e}} \Rightarrow z}{\text{listcase } e \text{ of } (e_{\text{e}}, e_{\text{ne}}) \Rightarrow z} \qquad \frac{e \Rightarrow z :: z' \qquad e_{\text{ne}}\, z\, z' \Rightarrow z''}{\text{listcase } e \text{ of } (e_{\text{e}}, e_{\text{ne}}) \Rightarrow z''}$$

Açúcar sintáctico

Embora possam ser definidas regras de avaliação para as definições e padrões, é mais simples ver essas construções como açúcar sintáctico.

- **Definições**

$$\text{let } p_1 \equiv e_1, \ldots, p_n \equiv e_n \text{ in } e \quad \doteq \quad (\lambda p_1. \ldots \lambda p_n.e)\, e_1 \ldots e_n$$

- **Padrões**

$$\lambda\langle p_1, \ldots, p_n\rangle.\, e \quad \doteq \quad \lambda v.\, \text{let } p_1 \equiv v.1, \ldots, p_n \equiv v.n \text{ in } e$$

Açúcar sintáctico

$$\text{let } \langle x, y\rangle \equiv w, z \equiv 2 * n \text{ in } x + y + z$$

$$\doteq \ (\lambda\langle x, y\rangle.\lambda z.x + y + z)\, w\, (2 * n)$$

$$\doteq \ (\lambda v.\, \text{let } x \equiv v.1, y \equiv v.2 \text{ in } \lambda z.x + y + z)\, w\, (2 * n)$$

$$\doteq \ (\lambda v.\, (\lambda x.\lambda y.\lambda z.\, x + y + z)\, (v.1)\, (v.2))\, w\, (2 * n)$$

Açúcar sintáctico

Uma definição alternativa para listas

- As listas podem ser vistas como estruturas de dados construidas à custa das alternativas e dos tuplos.
- Nesta abordagem uma lista é um valor alternativo que pode ser:
  - a etiqueta 1 seguida do tuplo vazio; ou
  - a etiqueta 2 seguida do par com a cabeça e cauda da lista.

$$\begin{array}{rcl} \mathsf{nil} & \doteq & @1\,\langle\rangle \\ e :: e' & \doteq & @2\,\langle e, e'\rangle \\ \mathsf{listcase}\ e\ \mathsf{of}\ (e_{\mathsf{e}}, e_{\mathsf{ne}}) & \doteq & \mathsf{sumcase}\ e\ \mathsf{of}\ (\lambda\langle\rangle.e_{\mathsf{e}}, \lambda\langle x,y\rangle.e_{\mathsf{ne}}\, x\, y) \\ & & \qquad\qquad \text{com } x, y \notin \mathsf{FV}(e_{\mathsf{ne}}) \end{array}$$

Recursividade

- Numa definição $\mathsf{let}\ v \equiv e\ \mathsf{in}\ e'$ as ocorrências de v em e' são ligadas, mas eventuais ocorrências de v em e são livres, dado que

$$\mathsf{let}\ v \equiv e\ \mathsf{in}\ e' \doteq (\lambda v.\, e')\, e$$

- Ou seja

$$\mathsf{FV}(\lambda p.\, e) = \mathsf{FV}(e) - \mathsf{FV}(p)$$

$$\begin{array}{l} \mathsf{FV}(\mathsf{let}\ p_1 \equiv e_1, \ldots, p_n \equiv e_n\ \mathsf{in}\ e) \\ \quad = \mathsf{FV}(e_1) \cup \ldots \cup \mathsf{FV}(e_n) \cup (\mathsf{FV}(e) - (\mathsf{FV}(p_1) \cup \ldots \cup \mathsf{FV}(p_n))) \end{array}$$

- Portanto, numa definição $\mathsf{let}\ v \equiv e\ \mathsf{in}\ e'$, v não pode representar uma função recursiva.

Recursividade

- A definição de funções recursivas é feita através da construção

$$\mathsf{letrec}\ \langle var\rangle \equiv \lambda\langle pat\rangle.\langle exp\rangle, \ldots, \langle var\rangle \equiv \lambda\langle pat\rangle.\langle exp\rangle\ \mathsf{in}\ \langle exp\rangle$$

  Note como as retrições sintáticas apenas deixam definir recursivamente funções (isto é característico do CBV).

- Assim,

$$\begin{array}{l} \mathsf{FV}(\mathsf{letrec}\ v_1 \equiv \lambda p_1.e_1, \ldots, v_n \equiv \lambda p_n.e_n\ \mathsf{in}\ e) \\ \quad = (\mathsf{FV}(\lambda p_1.e_1) \cup \ldots \cup \mathsf{FV}(\lambda p_n.e_n) \cup \mathsf{FV}(e)) - \{v_1, \ldots, v_n\} \end{array}$$

Semântica de avaliação "call-by-value"

- **Recursividade**

$$\frac{(\lambda v_1.\, \ldots \lambda v_n.\, e)\,(\lambda u_1.\, e_1^*) \cdots (\lambda u_n.\, e_n^*) \Rightarrow z}{\mathsf{letrec}\ v_1 \equiv \lambda u_1.e_1, \ldots, v_n \equiv \lambda u_n.e_n\ \mathsf{in}\ e \Rightarrow z}$$

  onde cada e_i^* representa $\mathsf{letrec}\ v_1 \equiv \lambda u_1.e_1, \ldots, v_n \equiv \lambda u_n.e_n\ \mathsf{in}\ e_i$ e $v_1, \ldots, v_n \notin \{u_1, \ldots, u_n\}$

Note que $(\lambda u_i.\, e_i^*)$ são formas canónicas. Portanto, os letrec's interiores não serão avaliados a não ser que a aplicação de $(\lambda u_i.\, e_i^*)$ a um argumento seja avaliada, ou seja, a não ser que a chamada recursiva seja avaliada.

Podemos ver $\mathsf{letrec}\ v_1 \equiv \lambda u_1.e_1, \ldots, v_n \equiv \lambda u_n.e_n\ \mathsf{in}\ e \Rightarrow z$ como a avaliação CBV da expressão e no contexto contendo as definições das funções $v_1, \ldots, v_n$, mutuamente recursivas.

Exemplos

Alguns exemplos

- A função factorial

$$\text{letrec fact} \equiv \lambda n.\, \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}\,(n-1) \ \ldots$$

- A concatenação de listas

$$\text{letrec append} \equiv \lambda x.\, \lambda y.\, \text{listcase } x \text{ of } (y, \lambda h.\, \lambda t.\, h :: \text{append}\, t\, y) \ \ldots$$

- A função *map*

$$\text{letrec map} \equiv \lambda f.\, \lambda l.\, \text{listcase } l \text{ of } (\text{nil}, \lambda h.\, \lambda t.\, f\, h :: \text{map}\, f\, t) \ \ldots$$

- A função *foldr*

$$\text{letrec foldr} \equiv \lambda f.\, \lambda z.\, \lambda l.\, \text{listcase } l \text{ of } (z, \lambda h.\, \lambda t.\, f\, h\, (\text{foldr}\, f\, z\, t)) \ \ldots$$

Exercícios

Apresente a avaliação de letrec fact $\equiv \ldots$ in $(\text{fact}\, 1)$ até à sua forma canónica.

Tendo definido funções de ordem superior, podemos definir novas funções utilizando definições não recursivas. Por exemplo:

$$\text{let append} \equiv \lambda x.\, \lambda y.\, \text{foldr}\,(\lambda h.\, \lambda r.\, h :: r)\, y\, x \ \ldots$$
$$\text{let inc} \equiv \lambda l.\, \text{map}\,(\lambda x.\, x + 1)\, l \ \ldots$$

Apresente definições alternativas para as funções fact e map.

Defina as seguintes funções (com recursividade explícita)

- Valor absoluto de um inteiro.
- Comprimento de uma lista.
- Testar se uma lista de inteiros está ordenada.
- Fusão de listas ordenadas.

O Sistema de Tipos

Tipos

Apresentamos agora um sistema de tipos simples para a nossa linguagem funcional.

- Sintaxe abstracta

$$\begin{array}{rcl}
\langle type\rangle & ::= & \mathbf{Int} \mid \mathbf{Bool} \mid \langle type\rangle \to \langle type\rangle \\
& \mid & \mathbf{Prod}(\langle type\rangle, \ldots, \langle type\rangle) \\
& \mid & \mathbf{Sum}(\langle type\rangle, \ldots, \langle type\rangle) \\
& \mid & \mathbf{List}\ \langle type\rangle
\end{array}$$

- Abreviaturas

$$\begin{array}{rcl}
\langle type\rangle \times \ldots \times \langle type\rangle & \doteq & \mathbf{Prod}(\langle type\rangle, \ldots, \langle type\rangle) \\
\mathbf{Unit} & \doteq & \mathbf{Prod}() \\
\langle type\rangle + \ldots + \langle type\rangle & \doteq & \mathbf{Sum}(\langle type\rangle, \ldots, \langle type\rangle)
\end{array}$$

- Precedências: $\mathbf{List}$, $\times$, $+$, $\to$
- Associatividade: $\times$, $+$ à esquerda; $\to$ à direita

Contextos

Para definir a relação de tipificação entre termos e tipos precisamos de introduzir a noção de *contexto* para declarar o tipo das variáveis (e dos padrões).

- Um contexto é uma lista de associações de tipos a padrões

$$\langle context \rangle \ ::= \ \mid \ \langle context \rangle, \langle pat \rangle : \langle type \rangle$$

com a restrição de uma variável não poder ocorrer mais do que uma vez num contexto.

- Um juizo de tipo tem a forma

$$\langle context \rangle \vdash \langle exp \rangle : \langle type \rangle$$

Sistema de tipos

- Definimos agora um sistema de tipos com base num conjunto de regras de inferência de tipos que especificam os juizos de tipos válidos.
- Usaremos as seguintes meta-variáveis

$$\begin{array}{llll} \Gamma & \langle context \rangle & x & \langle var \rangle \\ \theta, \tau, \sigma & \langle type \rangle & p & \langle pat \rangle \\ e & \langle exp \rangle & k, n & \mathbb{N} \end{array}$$

Regras de inferência de tipos

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Gamma, p : \sigma \vdash e : \tau}{\Gamma \vdash \lambda p.e : \sigma \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau}$$

Regras de inferência de tipos

$$\frac{}{\Gamma \vdash n : \mathbf{Int}} \ \text{para cada } n \in \mathbf{N}$$

$$\frac{\Gamma \vdash e : \mathbf{Int}}{\Gamma \vdash -e : \mathbf{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Int} \qquad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1 \ \mathbf{bop} \ e_2 : \mathbf{Int}} \ \text{para } \mathbf{bop} \in \{+, -, *, \mathsf{div}, \mathsf{mod}\}$$

Regras de inferência de tipos

$$\frac{}{\Gamma \vdash \mathsf{True} : \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{False} : \mathbf{Bool}}$$

$$\frac{\Gamma \vdash e : \mathbf{Bool}}{\Gamma \vdash \neg e : \mathbf{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Bool} \qquad \Gamma \vdash e_2 : \mathbf{Bool}}{\Gamma \vdash e_1 \ \mathbf{bop}\ e_2 : \mathbf{Bool}} \ \text{para } \mathbf{bop} \in \{\vee, \wedge\}$$

Regras de inferência de tipos

$$\frac{\Gamma \vdash e_1 : \mathbf{Int} \qquad \Gamma \vdash e_2 : \mathbf{Int}}{\Gamma \vdash e_1 \ \mathbf{bop}\ e_2 : \mathbf{Bool}} \ \text{para } \mathbf{bop} \in \{=, \neq, <, >, \leq, \geq\}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{Bool} \qquad \Gamma \vdash e_2 : \theta \qquad \Gamma \vdash e_3 : \theta}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \theta}$$

Regras de inferência de tipos

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{Unit}}$$

$$\frac{\Gamma \vdash e_1 : \theta_1 \ \cdots \ \Gamma \vdash e_n : \theta_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle : \theta_1 \times \ldots \times \theta_n}$$

$$\frac{\Gamma \vdash e : \theta_1 \times \ldots \times \theta_n}{\Gamma \vdash e.k : \theta_k} \ \text{para } k \in \{1, \ldots, n\}$$

Regras de inferência de tipos

$$\frac{\Gamma \vdash e : \theta_k}{\Gamma \vdash @k\, e : \theta_1 + \ldots + \theta_n} \ \text{para } k \in \{1, \ldots, n\}$$

$$\frac{\Gamma \vdash e : \theta_1 + \ldots + \theta_n \qquad \Gamma \vdash e_1 : \theta_1 \to \theta \ \cdots \ \Gamma \vdash e_n : \theta_n \to \theta}{\Gamma \vdash \mathsf{sumcase}\ e\ \mathsf{of}\ (e_1, \ldots, e_n) : \theta}$$

Regras de inferência de tipos

$$\frac{}{\Gamma \vdash \mathsf{nil} : \textbf{List}\ \theta} \qquad \frac{\Gamma \vdash e_1 : \theta \qquad \Gamma \vdash e_2 : \textbf{List}\ \theta}{\Gamma \vdash (e_1 :: e_2) : \textbf{List}\ \theta}$$

$$\frac{\Gamma \vdash e : \textbf{List}\ \theta \qquad \Gamma \vdash e_1 : \theta' \qquad \Gamma \vdash e_2 : \theta \to \textbf{List}\ \theta \to \theta'}{\Gamma \vdash \mathsf{listcase}\ e\ \mathsf{of}\ (e_1, e_2) : \theta'}$$

Regras de inferência de tipos

$$\frac{\Gamma, p_1 : \theta_1, \ldots, p_n : \theta_n \vdash e : \theta}{\Gamma, \langle p_1, \ldots, p_n \rangle : \theta_1 \times \ldots \times \theta_n \vdash e : \theta}$$

$$\frac{\Gamma \vdash e_1 : \theta_1 \ \ \ldots\ \ \Gamma \vdash e_n : \theta_n \qquad \Gamma, p_1 : \theta_1, \ldots, p_n : \theta_n \vdash e : \theta}{\Gamma \vdash \mathsf{let}\ p_1 \equiv e_1, \ldots, p_n \equiv e_n\ \mathsf{in}\ e : \theta}$$

Regras de inferência de tipos

$$\frac{\Gamma' \vdash e_1 : \theta_1 \ \ \ldots\ \ \Gamma' \vdash e_n : \theta_n \qquad \Gamma' \vdash e : \theta}{\Gamma \vdash \mathsf{letrec}\ p_1 \equiv e_1, \ldots, p_n \equiv e_n\ \mathsf{in}\ e : \theta} \quad \text{com } \Gamma' = \Gamma, p_1 : \theta_1, \ldots, p_n : \theta_n$$

Exercícios

Considere a seguinte expressão FACT

$$\mathsf{letrec\ fact} \equiv \lambda n.\, \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * \mathsf{fact}\,(n-1)\ \mathsf{in\ fact}$$

Construa uma árvore de prova do juizo

$$\vdash \mathsf{FACT} : \textbf{Int} \to \textbf{Int}$$

Considere a seguinte expressão APP

$$\mathsf{letrec\ append} \equiv \lambda x.\, \lambda y.\, \mathsf{listcase}\ x\ \mathsf{of}\ (y, \lambda h.\, \lambda t.\, h :: \mathsf{append}\, t\, y)$$
$$\mathsf{in\ append}\,(1 :: \mathsf{nil})\,(2 :: 7 :: 8 :: \mathsf{nil})$$

Construa uma árvore de prova do juizo

$$\vdash \mathsf{APP} : \textbf{List Int}$$

Exercícios

Construa uma extensão da linguagem de programação funcional, por forma a incluír um tipo de árvores binárias (com números inteiros nos nós).

Defina sintaxe abstracta, regras de inferência de tipos, e regras de avaliação CBV apropriadas.

As árvores binárias podem ser vistas como estruturas de dados construidas à custa das alternativas e dos tuplos, vendo os seus construtores e eliminadores como açucar sintáctico.

Apresente esta definição alternativa.

Uma Linguagem Funcional "call-by-name"

Uma linguagem funcional "call-by-name"

- Veremos agora uma pequena linguagem funcional com uma semântica que segue a ordem normal de avaliação. Ou seja, uma semântica "call-by-name" (CBN).
- A sintaxe da linguagem é idêntica à que vimos para a linguagem CBV, excepto para o caso das definições recursivas.
- A semântica CBN permite a formulação de um operador de ponto fixo

$$\langle exp \rangle ::= \mathsf{rec}\ \langle exp \rangle$$

- A semântica da linguagem CBN é bastante diferente da CBV: a avaliação dos operandos das aplicações, dos componentes dos tuplos e das alternativas é adiada até que seja evidente que esses valores são necessários para determinar o resultado do programa.

Uma linguagem funcional "call-by-name"

- O sistema de tipos para esta linguagem CBN é idêntico ao que vimos para a linguagem CBV. Apenas se acrescenta a seguinte regra de inferência de tipos

$$\frac{\Gamma \vdash e : \theta \to \theta}{\Gamma \vdash \mathsf{rec}\ e : \theta}$$

- Há expressões cuja avaliação CBN termina, embora possam divergir com uma avaliação CBV.
- Será possível definir estruturas de dados infinitas (como por exemplo as "lazy lists").

Formas canónicas

- A linguagem usa uma semântica de avaliação "call-by-name".
- $\Rightarrow$ relaciona as expressões com as formas canónicas da linguagem

$$\langle exp \rangle \Rightarrow \langle cfm \rangle$$

- Teremos formas canónicas de diferentes tipos

$$\begin{array}{lcl} \langle cfm \rangle & ::= & \lambda \langle var \rangle . \langle exp \rangle \\ & | & \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\ & | & \mathsf{True} \mid \mathsf{False} \\ & | & \langle \langle exp \rangle, \ldots, \langle exp \rangle \rangle \\ & | & @\langle tag \rangle \langle exp \rangle \\ & | & \mathsf{nil} \mid \langle exp \rangle :: \langle exp \rangle \end{array}$$

Semântica de avaliação "call-by-name"

- **Formas canónicas**

$$\frac{}{z \Rightarrow z}$$

- **Aplicação**

$$\frac{e \Rightarrow \lambda v.\widehat{e} \qquad \widehat{e}[e'/v] \Rightarrow z}{e\, e' \Rightarrow z}$$

Semântica de avaliação "call-by-name"

- **Operadores unários**

$$\frac{e \Rightarrow \lfloor i \rfloor}{-e \Rightarrow \lfloor -i \rfloor}$$

- **Operadores binários**

$$\frac{e_1 \Rightarrow \lfloor i_1 \rfloor \qquad e_2 \Rightarrow \lfloor i_2 \rfloor}{e_1 \textbf{ bop } e_2 \Rightarrow \lfloor i_1 \textbf{ bop } i_2 \rfloor} \text{ para } \textbf{bop} \in \{+,-,*,=,\neq,<,>,\leq,\geq\}$$

$$\frac{e_1 \Rightarrow \lfloor i_1 \rfloor \qquad e_2 \Rightarrow \lfloor i_2 \rfloor}{e_1 \textbf{ bop } e_2 \Rightarrow \lfloor i_1 \textbf{ bop } i_2 \rfloor} \text{ para } \textbf{bop} \in \{\mathsf{div}, \mathsf{mod}\} \text{ e } i_2 \neq 0$$

Açúcar sintáctico

- **Operadores booleanos**

$$\begin{array}{rcl} \neg e & \doteq & \mathsf{if}\ e\ \mathsf{then\ False\ else\ True} \\ e \lor e' & \doteq & \mathsf{if}\ e\ \mathsf{then\ True\ else}\ e' \\ e \land e' & \doteq & \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else\ False} \end{array}$$

Desta forma a avaliação de $\lor$ e $\land$ é em "curto-circuito".

Semântica de avaliação "call-by-name"

- **Expressões condicionais**

$$\frac{e_1 \Rightarrow \mathsf{True} \qquad e_2 \Rightarrow z}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow z}$$

$$\frac{e_1 \Rightarrow \mathsf{False} \qquad e_3 \Rightarrow z}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow z}$$

Semântica de avaliação "call-by-name"

- **Tuplos**

$$\frac{e \Rightarrow \langle e_1, \ldots, e_n \rangle \qquad e_k \Rightarrow z}{e.k \Rightarrow z} \ \text{para } k \in \{1, \ldots, n\}$$

Semântica de avaliação "call-by-name"

- **Alternativas**

$$\frac{e \Rightarrow @k\, e' \qquad e_k\, e' \Rightarrow z}{\mathsf{sumcase}\ e\ \mathsf{of}\ (e_1, \ldots, e_n) \Rightarrow z} \ \text{para } k \in \{1, \ldots, n\}$$

Semântica de avaliação "call-by-name"

- **Listas**

$$\frac{e \Rightarrow \mathsf{nil} \qquad e_{\mathsf{e}} \Rightarrow z}{\mathsf{listcase}\ e\ \mathsf{of}\ (e_{\mathsf{e}}, e_{\mathsf{ne}}) \Rightarrow z}$$

$$\frac{e \Rightarrow e' :: e'' \qquad e_{\mathsf{ne}}\, e'\, e'' \Rightarrow z}{\mathsf{listcase}\ e\ \mathsf{of}\ (e_{\mathsf{e}}, e_{\mathsf{ne}}) \Rightarrow z}$$

Semântica de avaliação "call-by-name"

- **Recursividade**

$$\frac{e\,(\mathsf{rec}\,e) \Rightarrow z}{\mathsf{rec}\,e \Rightarrow z}$$

Açúcar sintáctico

- Padrões, definições "let" e listas podem ser definidas como açúcar sintáctico do mesmo modo que na linguagem CBV.
- O operador de ponto fixo pode ser usado para definições "letrec" mais gerais do que tinhamos na linguagem CBV

$$\langle exp \rangle ::= \mathsf{letrec}\ \langle pat \rangle \equiv \langle exp \rangle, \ldots, \langle pat \rangle \equiv \langle exp \rangle\ \mathsf{in}\ \langle exp \rangle$$

- Onde

$$\begin{array}{l} \mathsf{FV}(\mathsf{letrec}\ p_1 \equiv e_1, \ldots, p_n \equiv e_n\ \mathsf{in}\ e) \\ \quad = (\mathsf{FV}(e_1) \cup \ldots \cup \mathsf{FV}(e_n) \cup \mathsf{FV}(e)) - \mathsf{Vars}(p_1) \ldots - \mathsf{Vars}(p_n) \end{array}$$

Açúcar sintáctico

- **Letrec**

$$\mathsf{letrec}\ p_1 \equiv e_1\ \mathsf{in}\ e \quad \doteq \quad \mathsf{let}\ p_1 \equiv \mathsf{rec}\,(\lambda p_1.\, e_1)\ \mathsf{in}\ e$$

$$\begin{array}{l} \mathsf{letrec}\ p_1 \equiv e_1, \ldots, p_n \equiv e_n\ \mathsf{in}\ e \\ \qquad \doteq \quad \mathsf{let}\ \langle p_1, \ldots, p_n \rangle \equiv \mathsf{rec}\,(\lambda \langle p_1, \ldots, p_n \rangle.\, \langle e_1, \ldots, e_n \rangle)\ \mathsf{in}\ e \end{array}$$

Exemplo

Considere a seguinte função que testa se duas listas são iguais

$$\mathsf{eqlist} : \mathbf{List\ Int} \to \mathbf{List\ Int} \to \mathbf{Bool}$$

e compare a avaliação CBN e CBV do seguinte programa

$$\begin{array}{l} \mathsf{letrec\ eqlist} \equiv \lambda l_1.\, \lambda l_2. \\ \quad \mathsf{listcase}\ l_1\ \mathsf{of}\ (\\ \qquad \mathsf{listcase}\ l_2\ \mathsf{of}\ (\mathsf{True}, \lambda h_2.\lambda t_2.\, \mathsf{False}), \\ \qquad \lambda h_1.\lambda t_1.\, \mathsf{listcase}\ l_2\ \mathsf{of}\ (\mathsf{False}, \lambda h_2.\lambda t_2.\, h_1 = h_2\ \wedge\ \mathsf{eqlist}\, t_1\, t_2) \\ \quad) \\ \mathsf{in}\quad \mathsf{eqlist}\ (1 :: 2 :: 3 :: \mathsf{nil})\ (3 :: 2 :: 1 :: \mathsf{nil}) \end{array}$$

Estruturas de dados infinitas

A semântica CBN permite definir estruturas de dados infinitas, como por exemplo as seguintes *"lazy lists"*

A lista infinita de zeros

$$\text{letrec zerolist} \equiv 0 :: \text{zerolist} \ \ldots$$

ou simplesmente, $\text{rec}\,(\lambda l.0 :: l)$

A lista infinita de números naturais

$$\begin{array}{ll} \text{letrec} & \text{map} \equiv \lambda f.\, \lambda l.\, \text{listcase}\ l\ \text{of}\ (\text{nil}, \lambda h.\lambda t.f\ h :: \text{map}\ f\ t) \\ \text{in} & \text{rec}\,(\lambda l.\, 0 :: \text{map}\,(\lambda x.x + 1)\, l) \end{array}$$

Apresente uma definição alternativa para a lista de naturais chamada natlist.

Como se comporta a função eqlist com listas infinitas?