

## 3 Semântica Operacional e Sistemas de Tipos

Vamos definir a semântica operacional de uma série de linguagens no estilo conhecido por *semântica operacional estrutural* chamado também de *semântica operacional small-step*.

O material dessas notas de aulas foi elaborado com base nas notas de aula de Peter Sewell, Universidade de Cambridge (parte sobre as linguagens L1, L2 e L3) e no livro *Types and Programming Languages* de Benjamin Pierce (parte sobre exceções, subtipos e orientação a objetos).

### 3.1 A Linguagem L1

Programas em L1 *pertencem* ao conjunto de árvores de sintaxe abstrata definido pela gramática abstrata abaixo:

Sintaxe de L1 (14)	
$e$	$::= n \mid b \mid e_1 \text{ op } e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$ $\mid x$ $\mid e_1 \ e_2$ $\mid \text{fn } x:T \Rightarrow e$ $\mid \text{let } x:T = e_1 \text{ in } e_2 \text{ end}$ $\mid \text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}$
$T$	$::= \text{int} \mid \text{bool} \mid T_1 \rightarrow T_2$
onde	$n \in \text{conjunto de numerais inteiros}$ $b \in \{\text{true}, \text{false}\}$ $\text{op} \in \{+, -, *, \geq, \dots\}$

Na gramática acima:

- $x$  representa um elemento pertencente ao conjunto *Ident* de identificadores
- $\text{fn } x:T \Rightarrow e$  é uma função (sem nome)
- $e_1 \ e_2$  é a aplicação da expressão  $e_1$  a expressão  $e_2$
- $\text{let } x:T = e_1 \text{ in } e_2 \text{ end}$  é uma expressão que permite declarar identificadores
- $\text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}$  permite o uso da função recursiva  $f$  dentro da expressão  $e_2$ .
- note que o programador deve escrever informação de tipo em programas

Observe que, de acordo com a gramática abstrata acima, fazem parte do conjunto de árvores de sintaxe abstrata expressões sem sentido tais como  $10 + \text{false}$  ou seja, nem todo elemento do conjunto definido pela gramática abstrata acima é uma expressão L1.

#### Sintaxe de L1 (15)

- Note que as convenções abaixo só interessam quando as árvores de sintaxe abstrata forem linearizadas
  - Aplicação é associativa a esquerda, logo  $e_1 \ e_2 \ e_3$  é o mesmo que  $(e_1 \ e_2) \ e_3$
  - As setas em tipos função são associativas a direita, logo o tipo  $T_1 \rightarrow T_2 \rightarrow T_3$  é o mesmo que  $T_1 \rightarrow (T_2 \rightarrow T_3)$
  - **fn** se estende o mais a direita possível, logo **fn**  $x:\text{int} \Rightarrow x + x$  é o mesmo que **fn**  $x:\text{int} \Rightarrow (x + x)$

### 3.1.1 Semântica Operacional de L1

Vamos definir a semântica operacional *small step* de L1. Começamos pela definição dos valores e dos *erros de execução* da linguagem:

#### Valores e Erros de Execução (16)

- Valores são determinadas expressões (noção sintática)
- Os valores da linguagem L1 são dados pela seguinte gramática:

$$v ::= n \mid b \mid \text{fn } x:T \Rightarrow e$$

- Se  $e \not\rightarrow$  e a expressão  $e$  não é valor temos um *erro de execução* (noção semântica)

A relação de transição  $\rightarrow$  é definida como sendo a menor relação que satisfaz um conjunto de propriedades dadas pelas seguintes regras:

#### Semântica Operacional de Operações Básicas (17)

$$\frac{\llbracket n \rrbracket = \llbracket n_1 \rrbracket + \llbracket n_2 \rrbracket}{n_1 + n_2 \rightarrow n} \quad (\text{OP}+) \quad$$

$$\frac{\llbracket b \rrbracket = \llbracket n_1 \rrbracket \geq \llbracket n_2 \rrbracket}{n_1 \geq n_2 \rightarrow b} \quad (\text{OP}\geq) \quad$$

$$\frac{e_1 \longrightarrow e'_1}{e_1 \text{ op } e_2 \longrightarrow e'_1 \text{ op } e_2} \quad (\text{OP1})$$

$$\frac{e_2 \longrightarrow e'_2}{v \text{ op } e_2 \longrightarrow v \text{ op } e'_2} \quad (\text{OP2})$$

As regras OP1 e OP2 acima são regras de reescrita, e as regras OP+ e OP $\geq$  são regras de computação. Observe que as regras OP1 e OP2 especificam que a avaliação dos operandos é feita da esquerda para direita. Observe também o uso das meta-variáveis  $n_1$  e  $n_2$  nas regras OP+ e OP $\geq$ . Dessa forma as regras especificam que a computação de + e  $\geq$  se dará somente nos casos em que ambos operandos forem números inteiros, caso contrário temos um erro de execução (expressão que não é valor mas para a qual não há regra de transição).

#### Condicional (18)

$$\text{if true then } e_2 \text{ else } e_3 \longrightarrow e_2 \quad (\text{IF1})$$

$$\text{if false then } e_2 \text{ else } e_3 \longrightarrow e_3 \quad (\text{IF2})$$

$$\frac{e_1 \longrightarrow e'_1}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow \text{if } e'_1 \text{ then } e_2 \text{ else } e_3} \quad (\text{IF3})$$

As regras IF1 e IF2 acima são regras de computação para o condicional, e a regra IF3 é uma regra de reescrita.

### Substituição (19)

Para a definição das regras para aplicação será necessário definir a operação de *substituição*.

- A semântica da aplicação de função a argumento envolve substituir variável por valor no corpo da função
- a notação  $\{v/x\}e$  representa a expressão que resulta da substituição de todas as **ocorrências livres** de  $x$  em  $e$  por  $v$ .
- Exemplos:

$$\{3/x\}(x + x) \equiv (3 + 3)$$

$$\{3/x\}(\text{fn } x : \text{int} \Rightarrow x + y) x \equiv (\text{fn } x : \text{int} \Rightarrow x + y) 3$$

$$\{2/x\}(\text{fn } y : \text{int} \Rightarrow x + y) \equiv \text{fn } y : \text{int} \Rightarrow 2 + y$$

### Aplicação (20)

$$(\text{fn } x : T \Rightarrow e) v \longrightarrow \{v/x\}e \quad (\beta)$$

$$\frac{e_2 \longrightarrow e'_2}{v e_2 \longrightarrow v e'_2} \quad (\text{APP1})$$

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad (\text{APP2})$$

Segue abaixo a definição da operação de substituição. Note que a condição associada a aplicação da substituição a funções e a expressões *let* e *let rec* garante que (i) somente variáveis livres vão ser substituídas e que (ii) nenhuma variável livre ficará indevidamente ligada após a substituição.

$$\begin{array}{ll}
\{e/x\} x & = e \\
\{e/x\} y & = y \text{ (se } x \neq y) \\
\{e/x\} \text{fn } y:T \Rightarrow e' & = \text{fn } z:T \Rightarrow \{e/x\}\{z/y\}e' \\
& \text{se } x \neq y \text{ e } z \notin fv(e) \cup fv(e') \cup \{x, y\} \\
\{e/x\} \text{fn } x:T \Rightarrow e' & = \text{fn } x:T \Rightarrow e' \\
\{e/x\} (e_1 e_2) & = (\{e/x\}e_1) (\{e/x\}e_2) \\
\{e/x\} n & = n \\
\{e/x\} (e_1 \text{ op } e_2) & = \{e/x\}e_1 \text{ op } \{e/x\}e_2 \\
\{e/x\} (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) & = \text{if } \{e/x\}e_1 \text{ then } \{e/x\}e_2 \text{ else } \{e/x\}e_3 \\
\{e/x\} b & = b \\
\{e/x\} (\text{let } y:T = e_1 \text{ in } e_2 \text{ end}) & = \text{let } z:T = \{e/x\}e_1 \text{ in } \{e/x\}\{z/y\}e_2 \text{ end} \\
& \text{se } x \neq y \text{ e } z \notin fv(e) \cup fv(e_2) \cup \{x, y\} \\
\{e/x\} (\text{let } x:T = e_1 \text{ in } e_2 \text{ end}) & = \text{let } x:T = \{e/x\}e_1 \text{ in } e_2 \text{ end} \\
\{e/x\} (\text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}) & = \text{let rec } z:T_1 \rightarrow T_2 = \{e/x\}(\text{fn } y:T_1 \Rightarrow \{z/f\}e_1) \\
& \text{in } \{e/x\}\{z/f\}e_2 \text{ end} \\
& \text{se } x \neq f \text{ e } z \notin fv(e) \cup fv(e_2) \cup \{x, f\} \\
\{e/f\} (\text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}) & = \text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}
\end{array}$$

Na definição acima  $fv(e)$  é o conjunto de variáveis livres da expressão  $e$  (variáveis que ocorrem mas não são declaradas em  $e$ ).

$$\begin{array}{ll}
fv(x) & = \{x\} \\
fv(\text{fn } y:T \Rightarrow e) & = fv(e) - \{y\} \\
fv(e_1 e_2) & = fv(e_1) \cup fv(e_2) \\
fv(n) & = \{\} \\
fv(e_1 \text{ op } e_2) & = fv(e_1) \cup fv(e_2) \\
\dots & \dots \dots
\end{array}$$

**Exercício 15** Termine a definição de  $fv$  iniciada acima.

Observação: essa é a definição mais geral de substituição. Se assumirmos que a expressão  $e$  em uma substituição  $\{e/x\}$  não possui variáveis livres a sua definição pode ser feita de forma bem mais simples. Esse de fato é o caso para programas onde todas as variáveis devem ser declaradas.

**Exercício 16** Prove que nenhuma variável livre é introduzida ao longo dos passos de avaliação de um programa fonte onde todas as variáveis estão declaradas

**Exercício 17** Simplifique a definição de substituição acima e defina  $\{v/x\}e$  com  $fv(v) = \{\}$ .

A expressão `let` serve para introduzir um novo identificador no programa e também para delimitar o seu escopo.

#### Regras de redução (21)

$$\text{let } x:T = v \text{ in } e_2 \text{ end} \longrightarrow \{v/x\}e_2 \quad (\text{LET1})$$

$$\frac{e_1 \longrightarrow e'_1}{\text{let } x:T = e_1 \text{ in } e_2 \text{ end} \longrightarrow \text{let } x:T = e'_1 \text{ in } e_2 \text{ end}} \quad (\text{LET2})$$

### Funções recursivas (22)

A sintaxe

$$e ::= \dots \mid \text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end}$$

permite descrever uma função  $f$  que, durante a sua execução, pode se chamar recursivamente.

Note que a construção **let** comum não permitiria isso, pois o nome  $f$  estaria *livre* dentro do corpo da função.

A avaliação de expressões **let rec**, da mesma forma que o **let**, substitui todas as ocorrências de  $f$  dentro da expressão  $e_2$ . Contudo, ao invés de substituir  $f$  pela função  $(\text{fn } y:T_1 \Rightarrow e_1)$ , ele substitui por  $(\text{fn } y:T_1 \Rightarrow \text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_1 \text{ end})$ , que mantém o nome  $f$  preso em  $e_1$  se alguma chamada recursiva a  $f$  precisar ser feita.

$$\begin{aligned} & \text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} \\ & \longrightarrow \\ & \{(\text{fn } y:T_1 \Rightarrow \text{let rec } f:T_1 \rightarrow T_2 = (\text{fn } y:T_1 \Rightarrow e_1) \text{ in } e_1 \text{ end})/f\}e_2 \end{aligned} \quad (\text{LETREC})$$

Segue abaixo a definição da função fatorial e a sua chamada para calcular o fatorial de 5 (neste exemplo supomos que operadores para igualdade, multiplicação e para subtração foram adicionados a linguagem):

```
let rec fat : int -> int =
  (fn y:int => if y = 0 then 1 else y * fat (y-1))
in fat 5
end
```

A expressão **let** pode ser considerada como simples açúcar sintático:

$$\text{let } x:T = e_1 \text{ in } e_2 \text{ end} \equiv (\text{fn } x:T \Rightarrow e_2) e_1$$

### 3.1.2 Sistema de Tipos para L1

Observe que a gramática abstrata definida no slide (1) admite expressões cuja avaliação leva a erro de execução (um erro de execução aqui é representado pela impossibilidade de aplicar uma regra da semântica operacional para uma expressão que não é valor). Vamos agora ver um *sistema de tipos* para a linguagem L1. Este sistema de tipos especifica uma análise estática a ser feita sobre árvores de sintaxe abstrata. Somente expressões consideradas *bem tipadas* por essa análise serão avaliadas.

Um sistema de tipos deve ser definido em acordo com a semântica operacional, em outras palavras, uma expressão só deve ser considerada bem tipada pelas regras de um sistema de tipos se a sua avaliação, pelas regras da semântica operacional, não levar a erro de execução. Essa propriedade fundamental é conhecida como *segurança* do sistema de tipos em relação a semântica operacional.

#### Tipos para L1 (23)

- Premissas e conclusão das regras do sistema de tipo são da forma  $\Gamma \vdash e : T$ , lido “a expressão  $e$  é do tipo  $T$  dadas as informações a cerca do tipo de identificadores que são mantidas em  $\Gamma$ ” onde:
  - $\Gamma$  é um mapeamento finito de identificadores para seus tipos (ambiente de tipos). Denotamos por  $\varepsilon$  o ambiente vazio, e  $(\Gamma, x : T)$  a extensão/redefinição de  $\Gamma$  com a informação de que o tipo da variável  $x$  é  $T$ .
  - $e$  é uma expressão da linguagem, e
  - $T$  é um tipo pertencente ao conjunto definido pela seguinte gramática

$$T ::= \text{int} \mid \text{bool} \mid T \rightarrow T$$

O tipo  $T_1 \rightarrow T_2$  é o tipo de funções cujo argumento é do tipo  $T_1$  e resultado é do tipo  $T_2$

Seguem abaixo as regras do sistema de tipos. Há uma regra para cada cláusula da gramática de expressões

#### Valores e Operações Básicas (24)

$$\Gamma \vdash n : \text{int} \quad (\text{TINT})$$

$$\Gamma \vdash b : \text{bool} \quad (\text{TBOOL})$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \quad (\text{T+})$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \geq e_2 : \text{bool}} \quad (\text{T}\geq)$$

Observe pelas regras OP+ e OP $\geq$  que os operandos de + e de  $\geq$  devem ser do tipo inteiro. Por estas regras expressões tais como  $4 + \text{true}$  e  $\text{true} \geq (\text{fn } x : T \Rightarrow 2)$  são consideradas *mal tipadas*.

#### Condicional (25)

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \quad (\text{TIF})$$

- Pela regra acima, para o condicional ser bem tipado a expressão da parte **then** e a expressão da parte **else** devem ser do mesmo tipo
- Note que expressões tais como **if**  $5 + 3 \geq 2$  **then true else** 5, de acordo com a semântica operacional, não levam a erro de execução. Mesmo assim, não são consideradas bem tipadas pela regra de tipo acima. Podemos dizer que o sistema de tipos está sendo muito conservador e recusando mais expressões do que deveria
- Isso acontece pois o sistema de tipos especifica uma análise *estática* feita sobre a árvore de sintaxe abstrata, ou seja sem saber se o resultado da avaliação do condicional virá da avaliação da subexpressão da parte **then** ou da subexpressão da parte **else**. Para poder concluir sobre o tipo de toda a expressão é preciso portanto, exigir que o tipo de ambas subexpressões seja o mesmo.

#### Tipando Funções (26)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad (\text{TVAR})$$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \text{fn } x : T \Rightarrow e : T \rightarrow T'} \quad (\text{TFN})$$

$$\frac{\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 e_2 : T'} \quad (\text{TAPP})$$



## Tipando expressão let (27)

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma, x : T \vdash e_2 : T'}{\Gamma \vdash \text{let } x : T = e_1 \text{ in } e_2 \text{ end} : T'} \quad (\text{TLET})$$

A regra de tipos para *let rec* precisa garantir que o corpo da função recursiva está bem tipado considerando a informação do parâmetro da função  $x$  e a possibilidade de chamada recursiva a  $f$ . Além disso, a expressão  $e_2$  também precisa estar bem tipada, supondo que possamos realizar chamadas a  $f$ .

$$\frac{\Gamma, f : T_1 \rightarrow T_2, y : T_1 \vdash e_1 : T_2 \quad \Gamma, f : T_1 \rightarrow T_2 \vdash e_2 : T}{\Gamma \vdash \text{let rec } f : T_1 \rightarrow T_2 = (\text{fn } y : T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} : T} \quad (\text{TLETREC})$$

## 3.1.3 Propriedades de L1

## Propriedades (28)

O teorema abaixo expressa que a avaliação, **em um passo**, é determinística

**Teorema 2 (Determinismo)** Se  $e \rightarrow e'$  e se  $e \rightarrow e''$  então  $e' = e''$ .

**Prova.** Por indução na estrutura de  $e$ . ■

A partir do teorema acima concluímos que a avaliação de programas L1 é determinística.

Na seção anterior vimos que é fundamental que um sistema de tipos seja seguro em relação a semântica operacional da linguagem. A noção de segurança foi então explicada de maneira informal:

## Segurança (29)

- Um sistema de tipos é seguro se expressões consideradas bem tipadas pelas suas regras não levam a **erro de execução** quando avaliadas de acordo com as regras da semântica operacional
- Erro de execução ocorre quando temos um estado/expressão  $e$  ao qual nenhuma regra da operacional se aplica
- De maneira **informal** podemos resumir essa noção de segurança através do seguinte *slogan*:

$$\text{Se } \Gamma \vdash e : T \text{ então } e \not\rightarrow^* \text{ erro}$$

A técnica de prova mais utilizada para provar que um sistema de tipos é seguro é conhecida como *segurança sintática*. Ela consiste em realizar basicamente duas provas:

Segurança Sintática (30)
<ul style="list-style-type: none"> <li>• prova do <b>progresso de expressões bem tipadas</b>, ou seja, provar que, se uma expressão for bem tipada, e se ela não for um valor, sua avaliação pode progredir um passo pelas regras da semântica operacional. <i>Slogan</i>:  <math display="block">\text{Se } \Gamma \vdash e : T \text{ então } e \text{ é valor, ou existe } e' \text{ tal que } (e \longrightarrow e')</math></li> <li>• prova da <b>preservação</b>, após um passo da avaliação, do tipo de uma expressão, ou seja, se uma expressão bem tipada progride em um passo, a expressão resultante possui o mesmo tipo da expressão original. <i>Slogan</i>:  <math display="block">\text{Se } \Gamma \vdash e : T \text{ e } (e \longrightarrow e') \text{ então } \Gamma \vdash e' : T.</math></li> </ul>

Note que ambas as provas são necessárias para provar segurança, ou seja:

$$\text{Segurança} = \text{Progresso} + \text{Preservação}$$

Provar somente *progresso* não é suficiente para provar segurança. É preciso provar que a expressão que resulta da progressão em um passo de uma expressão o bem tipada também é bem tipada (ou seja, que a propriedade de ser bem tipado é preservada pela avaliação em um passo). Da mesma forma, provar somente *preservação* não é suficiente para provar segurança. É preciso provar que a expressão bem tipada que resulta da progressão em um passo da expressão original pode progredir (ou seja, é preciso provar progresso em um passo de expressões bem tipadas). Observe que os *slogans* acima capturam a *essência* de progresso e preservação válida para qualquer linguagem de programação. Seguem abaixo as formulações precisas de progresso e preservação *específicas* para a linguagem L1.

Progresso e Preservação para L1 (31)
<p><b>Teorema 3 (Progresso)</b> Se <math>\Gamma \vdash e : T</math> então (i) <math>e</math> é valor, ou (ii) existe <math>e'</math> tal que <math>(e \longrightarrow e')</math></p> <p><b>Prova.</b> Por indução na estrutura de <math>e</math>. <span style="float: right;">■</span></p> <p><b>Teorema 4 (Preservação)</b> Se <math>\Gamma \vdash e : T</math> e <math>(e \longrightarrow e')</math> então <math>\Gamma \vdash e' : T</math>.</p> <p><b>Prova.</b> Por indução na estrutura de <math>e</math>. <span style="float: right;">■</span></p>

Estamos interessados em saber se os dois problemas abaixo são decidíveis, ou seja, se existem algoritmos que os resolvem.

---

 Problemas algorítmicos (32)
 

---

- Problema da **Verificação de Tipos**: dados ambiente  $\Gamma$ , expressão  $e$  e tipo  $T$ , o julgamento de tipo  $\Gamma \vdash e : T$  é derivável usando as regras do sistema de tipos?
- Problema da **Tipabilidade**: dados ambiente  $\Gamma$  e expressão  $e$ , encontrar tipo  $T$  tal que  $\Gamma \vdash e : T$  é derivável de acordo com as regras do sistema de tipos

O problema da tipabilidade é, mais difícil do que o problema da verificação de tipos para sistemas de tipos de linguagens de programação (para L1 na verdade ambos são triviais). Dependendo do sistema de tipos, resolver o problema da tipabilidade requer algoritmos de *inferência de tipos* muitas vezes complicados. No caso da linguagem L1 há algoritmos simples para ambos os problemas.

Observe que, do ponto de vista prático, o problema da *verificação de tipos* para L1 não é interessante<sup>1</sup>. Já o problema da *tipabilidade* é relevante na prática para L1: dado um programa L1 queremos saber se ele é ou não bem tipado e, ser for, queremos saber qual é o seu tipo.

---

 Algoritmo de Inferência de Tipos (33)
 

---

**Teorema 5** Dados ambiente  $\Gamma$  e expressão  $e$ , existe algoritmo que resolve o problema da tipabilidade para L1.

**Prova.** Exibir um algoritmo (ver exercício abaixo) que tem como entrada um ambiente de tipo  $\Gamma$  e uma expressão  $e$  e provar que o algoritmo: (i) termina sempre sua execução, e (ii) retorna um tipo  $T$  se e somente se  $\Gamma \vdash e : T$  é derivável de acordo com o sistema de tipos para L1 ■

**Observação :** Embora a diferença entre os dois problemas acima (da *verificação* de tipos e *tipabilidade*) seja clara, é bem comum se referir ao programa que os resolve como sendo o *verificador* de tipos para a linguagem.

---

<sup>1</sup>fora alguns exercícios de verificação de tipos a serem feitos

