Capítulo 4

A linguagem L1

Nesse capítulos vamos definir a semântica operacional e o sistema de tipos de uma linguagem funcional pura, denominada L1.

4.1 Sintaxe e semântica informal de L1

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

Sintaxe de L1

$$
\begin{array}{rcl}
e, e_1, e_2 \ldots & \in & \mathsf{L1} \\
e & ::= & n \\
& \mid & b \\
& \mid & e_1 \ \mathit{op} \ e_2 \\
& \mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& \mid & x \\
& \mid & e_1\ e_2 \\
& \mid & \texttt{fn}\ x{:}T \Rightarrow e \\
& \mid & \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2 \\
& \mid & \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2 \\
\\
T, T_1, T_2 \ldots & \in & \mathsf{Types} \\
T & ::= & \mathsf{int} \mid \mathsf{bool} \mid T_1 \to T_2
\end{array}
$$

onde

$$
\begin{array}{rcl}
n & \in & \textit{conjunto de numerais inteiros} \\
b & \in & \{\texttt{true}, \texttt{false}\} \\
x & \in & \mathsf{Ident} \\
\mathit{op} & \in & \{+, -, *, <, \leq, =, !=, \geq, >\}
\end{array}
$$

Na gramática vista acima:

- x representa um elemento pertencente ao conjunto Ident de identificadores escolhidos pelo programador
- $\texttt{fn}\ x{:}T \Rightarrow e$ é uma função anônima, de um único argumento x e cujo corpo é a expressão e

- $e_1\ e_2$ é a invocação da expressão e_1 (como função) usando a expressão e_2 como argumento. Uma expressão com essa estrutura sintática é chamada de *aplicação*
- $\texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2$ é uma expressão que permite declarar um identificador x com um tipo T, associar a esse identificador o valor de uma expressão (e_1) e delimitar o seu *escopo*, ou seja, porção de um programa onde o identificado declarado existe
- $\texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2$ é semelhante a expressão anterior. A diferença é que essa expressão é usada exclusivamente para a declaração de uma função recursiva

A linguagem L1 é explicitamente tipada e requer que todos os identificadores de um programa sejam declarados e seus tipos informados. Ela também é estaticamente tipada.

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

Do ponto de vista de estratégia de avaliação, L1 possui uma semântica *call by value* da esquerda para a direita.

Convenções sobre sintaxe concreta

- As convenções abaixo só interessam para sintaxe concreta pois questões como precedência e associatividade **não interessam em árvores de sintaxe abstrata**
- Essa discussão só é introduzida aqui pois nas notas de aula, para facilitar legibilidade, estamos representando árvores de sintaxe abstrata em um formato linearizado
  - 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 $T_1 \to T_2 \to T_3$ é o mesmo que $T_1 \to (T_2 \to T_3)$
  - $\texttt{fn}$ se estende o mais a direita possível, logo $\texttt{fn}\ x{:}\texttt{int} \Rightarrow x{+}x$ é o mesmo que $\texttt{fn}\ x{:}\texttt{int} \Rightarrow (x{+}x)$

Antes de definir formalmente a semântica operacional de L1 é preciso ter uma boa compreensão a cerca da linguagem e tomar algumas decisões sobre o seu projeto.

4.2 Semântica Operacional *small step* de L1

Vamos definir a semântica operacional *small step* de L1. Começamos pela definição dos valores e das expressões que são *erro de execução* (chamadas também de expressões *presas*) da linguagem:

Valores e expressões *erros de execução*

- Os **valores** da linguagem L1 pertencem ao conjunto definido pela seguinte gramática:

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

- Em L1, se uma expressão e *não* é valor e se não existe e' tal que $e \longrightarrow e'$ temos que e é uma expressão **erro de execução** (ou também uma expressão *presa*)

Semântica Operacional de Operações Básicas

$$\frac{[\![\mathsf{n}]\!] = [\![\mathsf{n}_1]\!] + [\![\mathsf{n}_2]\!]}{\mathsf{n}_1 \ \mathtt{+}\ \mathsf{n}_2 \longrightarrow \mathsf{n}} \ (\text{E-OP}+) \qquad \frac{e_1 \longrightarrow e_1'}{e_1 \ \mathsf{op}\ e_2 \longrightarrow e_1' \ \mathsf{op}\ e_2} \ (\text{E-OP1})$$

$$\frac{[\![\mathsf{n}_1]\!] \geq [\![\mathsf{n}_2]\!]}{\mathsf{n}_1 \geq \mathsf{n}_2 \longrightarrow \mathtt{true}} \ (\text{E-OP}\geq\text{TRUE}) \qquad \frac{e_2 \longrightarrow e_2'}{v \ \mathsf{op}\ e_2 \longrightarrow v \ \mathsf{op}\ e_2'} \ (\text{E-OP2})$$

$$\frac{[\![\mathsf{n}_1]\!] < [\![\mathsf{n}_2]\!]}{\mathsf{n}_1 \geq \mathsf{n}_2 \longrightarrow \mathtt{false}} \ (\text{E-OP}\geq\text{FALSE})$$

As regras E-OP1 e E-OP2 acima são regras de reescrita, e as regras E-OP+, E-OP$\geq$TRUE e E-OP$\geq$FALSE são regras de computação. Observe que as regras E-OP1 e 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 E-OP+, E-OP$\geq$TRUE e E-OP$\geq$FALSE. 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. As regras de computação para as demais operações ($-$,$*$,$<$,$==$,...) são muito similares às regras para $+$ e $\geq$, e portanto não as apresentaremos explicitamente (embora assumamos que estejam presentes na especificação)

Condicional

$$\mathtt{if\ true\ then}\ e_2\ \mathtt{else}\ e_3 \longrightarrow e_2 \ (\text{E-IFTRUE}) \qquad \mathtt{if\ false\ then}\ e_2\ \mathtt{else}\ e_3 \longrightarrow e_3 \ (\text{E-IFFALSE})$$

$$\frac{e_1 \longrightarrow e_1'}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \longrightarrow \mathtt{if}\ e_1'\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3} \ (\text{E-IF})$$

As regras E-IFTRUE e E-IFFALSE acima são regras de computação para o condicional, e a regra E-IF é uma regra de reescrita.

A semântica da aplicação de função a argumento, e também de expressões `let` e `letrec`, 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 pelo valor v. Exemplos:

$$\{3/x\}(x + x) \equiv 3 + 3$$
$$\{3/x\}((\mathtt{fn}\ x : \mathsf{int} => x + y)\ x) \equiv (\mathtt{fn}\ x : \mathsf{int} => x + y)\ 3$$
$$\{2/x\}(\mathtt{fn}\ y : \mathsf{int} => x + y) \equiv \mathtt{fn}\ y : \mathsf{int} => 2 + y$$

Aplicação de função

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

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \ (\text{E-APP1}) \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'} \ (\text{E-APP2})$$

A expressão let serve para introduzir um novo identificador no programa e também para delimitar o seu escopo. A regra de computação para expressão let também faz uso de substituição.

Let - declaração de identificadores

$$\texttt{let}\ x{:}T = v\ \texttt{in}\ e_2 \quad \longrightarrow \quad \{v/x\}e_2 \tag{LET1}$$

$$\frac{e_1 \quad \longrightarrow \quad e_1'}{\texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2 \quad \longrightarrow \quad \texttt{let}\ x{:}T = e_1'\ \texttt{in}\ e_2} \tag{LET2}$$

Observe que uma expressão `let` pode ser considerada como um açúcar sintático e pode ser traduzida para uma aplicação da seguinte forma:

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

A sintaxe

$$e ::= \ldots \mid \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2$$

permite declarar 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, f, ao invés de ser substituido pela função $(\texttt{fn}\ y{:}T_1 \Rightarrow e_1)$, é substituido por $(\texttt{fn}\ y{:}T_1 \Rightarrow \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_1)$, que mantém o nome f preso em e_1 se alguma chamada recursiva a f precisar ser feita.

Let rec - Funções recursivas

$$\begin{gathered}\texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2 \\ \longrightarrow \\ \{(\texttt{fn}\ y{:}T_1 \Rightarrow \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_1)/f\}e_2\end{gathered} \tag{LETREC}$$

Segue abaixo a definição da função fatorial e a sua chamada para calcular o fatorial de 5:

```
let rec fat : int → int =
    fn x : int => if x = 0 then 1 else x * fat(x−1)
in fat(5)
```

Podemos introduzir um açúcar sintático que melhora a legibilidade da definição de funções recursivas. Neste acúcar sintático, a função fatorial pode ser definida (e chamada) da seguinte forma:

```
let rec fat(x : int) : int =
      if x = 0 then 1 else x * fat(x−1)
in fat(5)
```

Exercício 22. *Defina uma semântica call by name para L1. Em uma semântica call by name o argumento de uma função é avaliado somente se for necessário.*

Exercício 23. *Dê exemplo de programa em L1 para o qual as semânticas call by value e call by name diferem no resultado*

Exercício 24. *Escreva um programa em L1 com a definição da função e a aplicação da função a um argumento. Escreva duas versões da função: a primeira usando açúcar sintática e a segunda versão no núcleo da linguagem.*

Exercício 25. *Explique porque a sintaxe do let recursivo é definida na forma:*

$$\texttt{let rec } f{:}T_1 \to T_2 = (\texttt{fn } y{:}T_1 \Rightarrow e_1) \texttt{ in } e_2$$

e não na forma $\texttt{let rec } f{:}T = (e_1) \texttt{ in } e_2$

Na sequência temos a definição da **operação de substituição**. Note que as condições associadas a aplicação da substituição a funções e a expressões *let* e *let rec* garante que somente variáveis livres vão ser substituídas.

A definição abaixo assume que o valor v a ser colocada no lugar das ocorrências livres de variáveis **não possui variáveis livres**. Essa suposição de fato é confirmada se consideramos que um programa bem tipado não possui variáveis livres (variáveis não declaradas) e que nenhuma regra da semântica operacional introduz variáveis livres.

Substituição: $\{\_/\_\}\_ : \text{L1} \times \text{Ident} \times \text{L1} \to \text{L1}$

$$
\begin{array}{lcl}
\{v/x\}\ n & = & n \\
\{v/x\}\ b & = & b \\
\{v/x\}\ (e_1 \text{ op } e_2) & = & \{v/x\}e_1 \text{ op } \{v/x\}e_2 \\
\{v/x\}\ (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \texttt{if } \{v/x\}e_1 \texttt{ then } \{v/x\}e_2 \texttt{ else } \{v/x\}e_3 \\
\{v/x\}\ (e_1\ e_2) & = & \{v/x\}e_1\ \{v/x\}e_2 \\
\{v/x\}\ x & = & v \\
\{v/x\}\ y & = & y \quad (se\ x \neq y) \\
\{v/x\}\ (\texttt{fn } x : T \Rightarrow e) & = & \texttt{fn } x : T \Rightarrow e \\
\{v/x\}\ (\texttt{fn } y : T \Rightarrow e) & = & \texttt{fn } y : T \Rightarrow \{v/x\}e \quad (\text{se } x \neq y) \\
\{v/x\}(\texttt{let } x{:}T = e_1 \texttt{ in } e_2) & = & \texttt{let } x{:}T = \{v/x\}e_1 \texttt{ in } e_2 \\
\{v/x\}(\texttt{let } y{:}T = e_1 \texttt{ in } e_2) & = & \texttt{let } y{:}T = \{v/x\}e_1 \texttt{ in } \{v/x\}e_2 \quad (\text{se } x \neq y) \\
\{v/f\}(\texttt{let rec } f : T_1 \to T_2 = (\texttt{fn } y{:}T_1 \Rightarrow e_1) & = & \texttt{let rec } f : T_1 \to T_2 = (\texttt{fn } y{:}T_1 \Rightarrow e_1) \\
\quad \texttt{in } e_2) & & \quad \texttt{in } e_2 \\
\{v/x\}(\texttt{let rec } f : T_1 \to T_2 = (\texttt{fn } y{:}T_1 \Rightarrow e_1) & = & \texttt{let rec } f : T_1 \to T_2 = \{v/x\}(\texttt{fn } y{:}T_1 \Rightarrow e_1) \\
\quad \texttt{in } e_2) & & \quad \texttt{in } \{v/x\}\ e_2 \\
 & & \quad \text{se } x \neq f
\end{array}
$$

As variáveis livres de uma expressão e são as variáveis que ocorrem em e mas que não são declaradas em e. Abaixo segue a definição do conjunto $fv(e)$ das variáveis livres de e:

Variáveis livres

$$
\begin{array}{lcl}
fv(n) & = & \{\,\} \\
fv(b) & = & \{\,\} \\
fv(e_1 \text{ op } e_2) & = & fv(e_1) \cup fv(e_2) \\
fv(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & fv(e_1) \cup fv(e_2) \cup fv(e_3) \\
fv(e_1\ e_2) & = & fv(e_1) \cup fv(e_2) \\
fv(x) & = & \{x\} \\
fv(\texttt{fn } x : T \Rightarrow e) & = & fv(e) - \{x\} \\
fv(\texttt{let } x{:}T = e_1 \texttt{ in } e_2) & = & fv(e_1) \cup (fv(e_2) - \{x\}) \\
fv(\texttt{let rec } f : T_1 \to T_2 = (\texttt{fn } x{:}T_1 \Rightarrow e_1) \texttt{ in } e_2) & = & (fv(e_1) - \{x\}) \cup fv(e_2)) - \{f\}
\end{array}
$$

4.3 Sistema de Tipos para L1

Um sistema de tipos deve ser definido levando em consideração as regras da semântica operacional. 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. Somente expressões consideradas *bem tipadas* por essa análise serão avaliadas.

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* Γ".

O ambiente de tipos Γ captura a essência de uma *tabela de símbolos* de um compilador/interpretador formalizada aqui como uma função de identificadores declarados para seus tipos. A notação $\Gamma(x)$ pode ser vista como uma abstração para uma operação $\text{lookup}(\Gamma, x)$. E a notação $\Gamma, x : T$ (usada na regra para funções, `let`, e `let rec`), representa a operação $\text{update}(\Gamma, (x, T))$.

Os tipos para a linguagem L1 são definidos pela seguinte gramática abstrata, onde o tipo $T_1 \to T_2$ é o tipo de funções cujo argumento é do tipo T_1 e resultado é do tipo T_2

$$T \quad ::= \quad \text{int} \mid \text{bool} \mid T_1 \to T_2$$

Pelas regras E-OP+ e E-OP$\geq$ da semântica operacional, os operandos de $+$ e de $\geq$ devem ser do tipo inteiro. expressões tais como $4 + \texttt{true}$ e $\texttt{true} \geq (\texttt{fn } x : \text{int} \Rightarrow 2)$, por exemplo, são consideradas *mal tipadas*.

Regras de tipo para valores e operações básicas

$$\frac{}{\Gamma \vdash n : \text{int}} \ (\textsc{Tint}) \qquad \frac{}{\Gamma \vdash b : \text{bool}} \ (\textsc{Tbool})$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \ (\text{TOP}+) \qquad \frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \geq e_2 : \text{bool}} \ (\text{TOP}\geq)$$

Para o condicional ser bem tipado as expressões da parte `then` e da parte `else` devem ser do mesmo tipo.

Condicional

$$\frac{\Gamma \vdash e_1 : \text{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T} \ (\textsc{Tif})$$

Note que expressões tais como $\texttt{if}\ 5+3 \geq 2\ \texttt{then true else}\ 5$, 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 subexpresssão 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.

Um programa L1 para ser bem tipado não deve possuir variáveis livres, ou seja, L1 é uma linguagem de programação na qual todos os identificadores devem ser declarados. Na regra abaixo para tipar variáveis, a premissa é falsa no caso de $\Gamma(x)$ ser indefinido. Isso significa que o identificador x não foi declarado no programa.

Variáveis

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad (\textsc{Tvar})$$

Na premissa da regra $\textsc{Tfn}$ abaixo, para tipar o corpo da função, o ambiente deve deve ser antes atualizado com o nome e com o tipo do argumento da função, tal como informado pelo programador. Na regra $\textsc{App}$ para aplicação o tipo de entrada da função deve ser igual ao tipo da expressão sendo aplicada a ela.

Tipando funções e aplicação

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \texttt{fn}\ x : T \Rightarrow e :\ T \to T'} \quad (\textsc{Tfn}) \qquad \frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'} \quad (\textsc{App})$$

Tanto para as expressões let como let rec é importante acertar o ambiente de tipos nos quais as suas subexpressões serão tipadas. Na expressão let note que o identificador x é declarado para ser usado na subexpressão e_2.

Tipando expressão let

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma, x : T \vdash e_2 : T'}{\Gamma \vdash \texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2 : T'} \quad (\textsc{Tlet})$$

A regra de tipos para let rec precisa garantir que o corpo da função recursiva está bem tipado considerando a informação do argumento x da função 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.

Tipando expressão letrec

$$\frac{\Gamma,\ f{:}\ T_1 \to T_2,\ x : T_1 \vdash e_1 : T_2 \qquad \Gamma,\ f{:}\ T_1 \to T_2 \vdash e_2 : T}{\Gamma \vdash \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ x{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2 : T} \quad (\textsc{Tletrec})$$

Exercício 26. *Explique porque há duas construções do tipo let na linguagem (uma para definições não*

recursivas e outra exclusiva para definição de funções recursivas. Em outras palavras explique por que uma função como fatorial, por exemplo, não pode ser definida em L1 com `let` *dessa forma:*

```
let fat : int → int = fn x : int =
      if x = 0 then 1 else x * fat(x−1)
in fat(5)
```

Exercício 27. *Em um projeto de linguagem o ideal é que conceitos/construções sejam incorporados a linguagem de forma ortogonal, ou seja, interagindo da mesma forma com as demais construções da linguagem e sem casos excepcionais/específicos de uso. O conceito de funções é ortogonal na linguagem L1?*

Exercício 28. *Defina regras de tipo e também regras de semântica operacional para a igualdade (expressões na forma $e_1 == e_2$).*

4.4 Propriedades de L1

Com a linguagem L1 definida precisamente de forma indutiva sobre a gramática abstrata (usando lógica como meta-linguagem de definição), é possível **provar** propriedades a cerca da linguagem.

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

Teorema 12 (Determinismo)**.** *Se $e \longrightarrow e'$ e se $e \longrightarrow e''$ então $e' = e''$.*

Demonstração. Por indução na estrutura de e. □

Na seção anterior vimos que é fundamental que um sistema de tipos seja seguro em relação a semântica operacional da linguagem.

Segurança do Sistema de Tipos

- 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* o ocorre quando temos uma expressão e, não valor, a qual nenhuma regra da semântica operacional se aplica

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:

Para provar Segurança Sintática

- provar **progresso de expressões bem tipadas**, ou seja, provar que, se uma expressão for bem tipada, e se ela não for um valor, sua avaliação pode progredir em um passo pelas regras da semântica operacional.
- provar **preservação, 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.

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

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

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 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, caso não seja um valor (ou seja, é preciso provar progresso em um passo de expressões bem tipadas).

Seguem abaixo as formulações precisas de progresso e preservação *específicas* para a linguagem L1.

Progresso e Preservação para L1

Teorema 13 (Progresso)**.** *Se $\emptyset \vdash e : T$ então (i) e é valor, ou (ii) existe e' tal que $e \longrightarrow e'$*

Demonstração. Por indução na estrutura de e. □

Teorema 14 (Preservação)**.** *Se $\vdash e : T$ e $(e \longrightarrow e')$ então $\vdash e' : T$.*

Demonstração. Por indução na estrutura de e. □

4.5 Problemas algorítmicos

Até aqui a ênfase foi (i) na especificação das semânticas dinâmica (semântica operacional) e estática (sistema de tipos) da linguagem, e (ii) na relação entre essas dois sistemas formais. Estamos interessados em investigar os seguintes problemas algoritmicos sobre a tipagem de programas:

Problemas algorítmicos

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

Observe que, do ponto de vista prático, o problema da *verificação de tipos* para L1 não é interessante. Já o problema da *inferência de tipos* 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. O problema da inferência é mais difícil do que o problema da verificação de tipos para sistemas de tipos de linguagens de programação. No caso da linguagem L1 há algoritmos simples para ambos os problemas.

O algoritmo `typeInfer` está correto se ele satisfaz as seguintes propriedades:

- **Terminação** - `typeInfer` sempre **termina** sua execução, dada qualquer ambiente de tipo e expressão de $L1$ como entrada,
- **Segurança** - `typeInfer` é **seguro** em relação as regras do sistema de tipos para $L1$, ou seja, se $\texttt{typeInfer}(\Gamma, e) = T$ então $\Gamma \vdash e : T$, e
- **Completeza** `typeInfer` é **completo** em relação as regras do sistema de tipos para $L1$, ou seja, se $\Gamma \vdash e : T$ então $\texttt{typeInfer}(\Gamma, e) = T$.

Abaixo segue um algoritmo de **inferência** de tipos para $L1$ em uma pseudo linguagem algorítmica funcional:

```
typeInfer(Γ, e) =
case e
  true ⇒ bool (* T-True *)
| false ⇒ bool (* T-False *)
| n    ⇒  int (* T-Num *)
| e1 + e2 ⇒ se typeInfer(Γ, e1)=int ∧ typeInfer(Γ, e2)=int então int
             senão erro (* T-Op+ *)
| e1 e2 ⇒ se typeInfer(Γ, e1) = T1 → T2 ∧ typeInfer(Γ, e2) = T1' ∧ T1 = T1' então T2
           senão erro (* T-APP *)
| x ⇒ lookup(Γ, x) (*T-Var − lookpup retorna erro se x nao foi declarado *)
| fn x : T ⇒ e' ⇒ T → typeInfer(update(Γ, x, T), e') (* T-Fn *)
| if(e1, e2, e3) ⇒ se typeInfer(Γ, e1) = bool então (* T-If *)
                                {T2 = typeInfer(Γ, e2)
                                 T3 = typeInfer(Γ, e3)
                                 se T2 = T3 então T2 senão erro}
                   senão erro
 | let x : T = e1 in e2 ⇒ (* T-let *)
                     se typeInfer(Γ, e1) = T então typeInfer(update(Γ, x, T), e2)
                     senão erro
 | let rec f : T → T' = fn x : T ⇒ e1 in e2 ⇒ (* T-LetRec *)
                  Γ' = update(Γ, f, T → T')
                  se typeInfer(update(Γ', x, T), e1) = T' então typeInfer(Γ', e2)
                  senão erro
```

Teorema 15. *O algoritmo* `typeInfer` *é correto, ou seja ele sempre termina e é seguro e completo em relação ao sistema de tipos para* $L1$.

Prova. Segurança e completeza podem ser provadas por indução na estrutura da expressão de entrada e. Terminação segue do fato de que a única fonte possível de não terminação são as chamadas recursivas de `typeInfer`. Por inspeção no algoritmo vemos que a cada chamada de `typeInfer` o tamanho da AST de entrada diminui até que ele seja chamado com nodos folhas (n, `true`, `false`, x).

As regras do sistema de tipos especificam uma análise estática a ser incorporada em um interpretador ou em um compilador. O avaliador de expressões, no caso de um interpretador, ou o gerador de código, no caso de um compilador, só será executado se essa análise estática da AST feita por `typeInfer` concluir que ela é bem tipada.

Exercício 29. *Prove que* `typeInfer` *é correto.*

Exercício 30. *Modifique typeInfer para a L1 e mostre 3 versões, uma para cada caso abaixo*

1. *typeInfer seguro mas não completo*
2. *typeInfer completo, mas não seguro*
3. *typeInfer inseguro e incompleto.*

Exercício 31. *Implementar o algoritmo typeInfer para L1 na sua linguagem de programação favorita.*

4.6 Semântica operacional *big step*

A técnica mais usada para provar seguranca de sistemas de tipos, conhecida como **segurança sintática**, requer semântica operacional *small step*. Avaliadores seguindo o estilo *small step* são, porém, bastante ineficientes pois a **cada passo** da avaliação é preciso

1. percorrer a AST para localizar onde fazer a transformação
2. efetuar a transformação gerando uma AST modificada
3. efetuar substituição de identificador por AST de valor (caso o passo envolva substituição)

Avaliadores no estilo *big step* são mais eficientes. Há dois estilos de semântica operacional *big step*

- *big step* com substituição (elimina ineficiências (1) e (2) acima)
- *big step* com ambientes (ou seja, sem substituições)

4.6.1 Semântica operacional *big step* com substituição para $L1$

Consiste na definição de uma relação binária $\Downarrow: L1 \times \text{Values}$ onde Values é como definido para semântica operacional *small step*:

$$v \quad ::= \quad \ldots \mid \; n \mid b \mid fn\ x{:}T \Rightarrow e$$

Semântica Operacional *big step* com substituição para $L1$

$$\frac{}{n \Downarrow n}\ (\textsc{bnum}) \qquad \frac{}{b \Downarrow b}\ (\textsc{bbool})$$

$$\frac{e_1 \Downarrow \texttt{false} \qquad e_3 \Downarrow v}{\rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}\ (\textsc{biff}) \qquad \frac{e_1 \Downarrow \texttt{true} \qquad e_2 \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}\ (\textsc{bift})$$

$$\frac{}{\texttt{fn}\ x{:}T \Rightarrow e \Downarrow \texttt{fn}\ x{:}T \Rightarrow e}\ (\textsc{bfn}) \qquad \frac{e_1 \Downarrow v' \qquad \{v'/x\}\ e_2 \Downarrow v}{\texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2 \Downarrow v}\ (\textsc{blet})$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad [\![n]\!] = [\![n_1]\!] + [\![n_2]\!]}{e_1 + e_2 \Downarrow n}\ (\textsc{bop}+) \qquad \frac{\{\boldsymbol{\alpha}/f\}\ e_2 \Downarrow v}{\texttt{let rec}\ f{:}T \to T' = \texttt{fn}\ x{:}T \Rightarrow e_1\ \texttt{in}\ e_2 \Downarrow v}\ (\textsc{blrec})$$

$$\frac{e_1 \Downarrow \texttt{fn}\ x{:}T \Rightarrow e \qquad e_2 \Downarrow v_2 \qquad \{v_2/x\}\ e \Downarrow v}{e_1\ e_2 \Downarrow v}\ (\textsc{bapp})$$

$$\boldsymbol{\alpha} \quad \equiv \quad \texttt{fn}\ x{:}T \Rightarrow \texttt{let rec}\ \ f{:}T \to T' = (\texttt{fn}\ x{:}T \Rightarrow e_1)\ \texttt{in} \quad e_1$$

4.6.2 Semântica operacional *big step* com ambientes para L1

Nessa seção, será apresentada uma semântica *big step* para L1 que faz uso de *ambientes* e de *closures*. Nesse estilo de semântica, ao invés de realizar a substituição de identificadores por valores, é construída uma estrutura de dados (ambiente) que descreve a substituição. A substituição só é feita *sob demanda* ao nos depararmos com variáveis. Esse estilo reflete de forma mais realista como funções são implementadas por compiladores, e tende a ser mais eficiente que o modelo dado por substituição direta. Abaixo segue a definição de *ambientes* e *valores*.

Ambientes e valores para $L1$

$$\begin{array}{lcl@{\qquad\qquad}lcl} v & \in & \text{Values} & \rho & \in & \text{Env} \\ v & ::= & n \mid b \mid \langle x, e, \rho \rangle & \rho & ::= & [] \mid \rho, x \mapsto v \end{array}$$

Observação: nesta seção, para simplificar a apresentação serão omitidas informações de tipo presentes em expressões L1. Em particular, note que o tipo T do argumento da função é omitido na closure correspondente, dado que este é necessário para o sistema de tipos mas não para a semântica operacional.

Ao utilizarmos esse modelo, contudo, precisamos ser mais cuidadosos com questões de *escopo*. O valor $\langle x, e, \rho \rangle$ é denominada **closure** e representa uma função com seu argumento x, seu corpo e e o ambiente ρ. Esse ambiente ρ corresponde ao ambiente vigente no **momento que a função é avaliada como valor**. O *closure* carrega esse ambiente para dar sentido às *variáveis* que possam ocorrer dentro do corpo da função mas cujos valores foram definidos fora do corpo da função (chamadas de variáveis livres da função). Esse mecanismo é chamado de *escopo estático*.

Se isso não fosse feito, os valores das variáveis livres da função seriam obtidos no ambiente vigente no momento da chamada da função, o que é conhecido como *escopo dinâmico*. Escopo dinâmico foi implementado em versões iniciais da linguagem de programação LISP, e caiu em desuso por ser difícil de ser utilizado na prática de forma correta. Praticamente todas as linguagens de programação modernas (C++, Java, Pascal, Python, Ocaml, Haskell, Javascript, ...) utilizam *escopo estático*.

Para ilustrar a diferença entre escopo dinâmico e estático, considere o seguinte código em L1:

```
let x = 2 in
  let foo = (fn y => x+y) in
    let x = 5 in
      foo (10)
```

Sob *escopo estático*, a ocorrência de x dentro do corpo de foo deve avaliar sempre para 2, que é valor associado a x no momento da definição da função. Dessa forma, o código acima deve simplificar para 12. Sob *escopo dinâmico*, a interpretação de x seria a do escopo vigente no momento da chamada, no caso 5. Portanto, o código acima resultaria em 15.

Exercício 32. *Faça a avaliação do código acima utilizando a semântica small-step e responda: a semântica small-step (com substituição direta) para L1 implementa escopo estático ou escopo dinâmico?*

As premissas e conclusão das semântica *big step* para L1 tem o formato

$$\rho \vdash e \Downarrow v$$

que é lido como "a expressão e avalia para o valor v sob ambiente ρ". Começamos com uma versão reduzida de L1 sem expressão `let rec` para definição de funções recursivas. Os valores produzidos são números, booleamos e *closures* de funções. Note que o ambiente ρ é uma função de identificadores para valores definida como segue:

$$(\rho, x \mapsto v)(y) = \begin{cases} v & \text{se } x = y \\ \rho(x) & \text{caso contrario} \end{cases}$$

Semântica Operacional *big step* para $L1$

$$\rho \vdash n \Downarrow n \quad \text{(BS-Num)} \qquad \rho \vdash b \Downarrow b \quad \text{(BS-bool)}$$

$$\frac{\rho \vdash e_1 \Downarrow n_1 \qquad \rho \vdash e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{\rho \vdash e_1 \texttt{+}\ e_2 \Downarrow n} \quad \text{(BS-Sum)}$$

$$\frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \quad \text{(BS-Id)}$$

$$\frac{\rho \vdash e_1 \Downarrow \texttt{true} \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \text{(BS-IfTr)} \qquad \frac{\rho \vdash e_1 \Downarrow \texttt{false} \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \text{(BS-IfFls)}$$

$$\rho \vdash (\texttt{fn}\ x \Rightarrow e) \Downarrow \langle x, e, \rho \rangle \quad \text{(BS-Fn)}$$

$$\frac{\rho \vdash e_1 \Downarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \Downarrow v' \qquad \rho', x \mapsto v' \vdash e \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-App)}$$

$$\frac{\rho \vdash e_1 \Downarrow v' \qquad \rho, x \mapsto v' \vdash e_2 \Downarrow v}{\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v} \quad \text{(BS-Let)}$$

A regra BS-Id simplesmente consulta o ambiente atual para substituir uma dada variável x. A regra BS-Fn cria uma *closure* para uma dada função. Note que o ambiente armazenado na closure é exatamente o ambiente corrente no momento em que a função é avaliada. A regra BS-App especifica que, ao avaliar uma aplicação $e_1\ e_2$, a expressão e_1 deve avaliar para uma closure $\langle x, e, \rho' \rangle$. A terceira premissa dessa regra diz que o componente e da closure (o corpo da função) deve ser avaliado para produzir o valor resultante da aplicação. Essa avaliação se dá sob o ambiente ρ' contido na closure, mas estendido com a associação do parâmetro x da função com o valor v resultante da avaliação da expressão e_2 da aplicação.

Com funções recursivas definidas através `let rec` o conjunto de valores é aumentado com closures para funções recursivas. Na gramática abaixo, $\langle f, x, e, \rho \rangle$ é a closure recursiva onde f é o identificador da função recursiva, x é o argumento da função anônima associada a f, e é o corpo dessa função anônima, e ρ é o ambiente corrente quando da declaração da função recursiva no programa:

$$v \quad ::= \ldots \quad \mid \langle f, x, e, \rho \rangle$$

Ao aplicarmos uma closure recursiva, o ambiente na qual o corpo irá rodar é estendido com o parâmetro x e com a definição da própria closure recursiva, salva sob o nome f. Isto é, o nome f *preserva* a definição da closure para potenciais chamadas recursivas que ocorram dentro de e (corpo da closure).

Semântica operacional com `let rec`

$$\frac{\rho \vdash e_1 \Downarrow \langle f, x, e, \rho' \rangle \qquad \rho \vdash e_2 \Downarrow v' \qquad \rho',\ x \mapsto v',\ f \mapsto \langle f, x, e, \rho' \rangle \vdash e \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-AppRec)}$$

$$\frac{\rho, f \mapsto \langle f, x, e_1, \rho \rangle \vdash e_2 \Downarrow v}{\rho \vdash \texttt{let rec}\ f = \texttt{fn}\ x \Rightarrow e_1\ \texttt{in}\ e_2 \Downarrow v} \quad \text{(BS-LetRec)}$$

Capítulo 5

Extensões

Neste capítulo, vamos estudar como estender L1 com construções bastante comuns em linguagens modernas tais como registros, pares ordenados, listas, e exceções.

5.1 Registros e pares ordenados

Registros e pares são estruturas de dados heterogêneas, ou seja os elementos que os compõem podem ser de tipos diferentes. A gramática abaixo mostra as novas construções adicionadas a L1. Observar que a quantidade variável de componentes de um registro irá introduzir um grau de informalidade na especificação da sintaxe e nas regras da semântica operacional e do sistema de tipos.

$$
\begin{array}{rcll}
e & ::= & \ldots & \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
& & & \mid \{lab_1 = e_1, \ldots lab_n = e_n\} \mid \sharp lab\ e \qquad n \geq 0
\end{array}
$$

Os operadores unários `fst` e `snd` retornam o primeiro e segundo componente de um par ordenado. Note que *lab*, lab_1, ... pertencem a um conjunto de identificadores Lab para rótulos de componentes de registros. Rótulos não podem rer repetidos em um mesmo registro. A operação $\sharp lab$, aplicada a um registro projeta seu componente de rótulo lab.

A sintaxe de registros é muito semelhante a sintaxe de tipos de registros. Nós temos, por exemplo, que o registro $\{A = 5, B = \texttt{true}, C = \{D = 20, E = 40\}\}$ é do tipo $\{A : \mathsf{int}, B : \mathsf{bool}, C : \{D : \mathsf{int}, E : \mathsf{int}\}\}$.

$$
\begin{array}{rcll}
T & ::= & \ldots & \mid T_1 * T_2 \\
& & & \mid \{lab_1 : T_1, \ldots lab_n : T_n\} \qquad n \geq 0
\end{array}
$$

Pares de valores e registros cujos componentes são todos valores, são também valores.

$$
\begin{array}{rcll}
v & ::= & \ldots & \mid (v_1, v_2) \\
& & & \mid \{lab_1 = v_1, \ldots lab_n = v_n\} \qquad n \geq 0
\end{array}
$$

Os componente de pares e registros são avaliados da esquerda para a direita. As regras da semântica operacional *small step* para pares são triviais:

$$\frac{e_1 \longrightarrow e_1'}{(e_1, e_2) \longrightarrow (e_1', e_2)}\ (\textsc{par1}) \qquad \frac{e \longrightarrow e'}{\texttt{fst}\ e \longrightarrow \texttt{fst}\ e'}\ (\textsc{prj1}) \qquad \frac{}{\texttt{fst}\ (v_1, v_2) \longrightarrow v_1}\ (\textsc{prj1v})$$

$$\frac{e_2 \longrightarrow e_2'}{(v, e_2) \longrightarrow (v, e_2')}\ (\textsc{par2}) \qquad \frac{e \longrightarrow e'}{\texttt{snd}\ e \longrightarrow \texttt{snd}\ e'}\ (\textsc{prj2}) \qquad \frac{}{\texttt{snd}\ (v_1, v_2) \longrightarrow v_2}\ (\textsc{prj2v})$$

As regras para registros são conceitualmente simples: para avaliar um registro basta avaliar seus componentes da esquerda para a direita. Há uma dificuldade notacional devido ao tamanho variável de registros.

$$\frac{e_j \longrightarrow e_j'}{\{lab_i = v_i^{\,i \in 1\ldots j-1},\ lab_j = e_j,\ lab_k = e_k^{\,k \in j+1\ldots n}\} \longrightarrow \{lab_i = v_i^{\,i \in 1\ldots j-1},\ lab_j = e_j',\ lab_k = e_k^{\,k \in j+1\ldots n}\}}\ (\textsc{rcd})$$

$$\frac{e \longrightarrow e'}{\sharp lab_i\ e \longrightarrow \sharp lab_i\ e'}\ (\textsc{prjrd}) \qquad \frac{}{\sharp lab_i\ \{lab_1 = v_1, \ldots lab_n = v_n\} \longrightarrow v_i}\ (\textsc{prjrdv})$$

As regras de tipos para pares ordenados e operações com pares ordenados são triviais Note que os pares $(e_1, (e_2, e_3))$ e $((e_1, e_2), e_3)$ são de tipos diferentes!

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 * T_2}\ (\textsc{Tpar}) \qquad \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \texttt{fst}\ e : T_1}\ (\textsc{Tprj1}) \qquad \frac{\Gamma \vdash e : T_1 * T_2}{\Gamma \vdash \texttt{snd}\ e : T_2}\ (\textsc{Tprj2})$$

Tendo em vista que as regras para registro não são estritamente formais, e para não carregar mais a notação, a regra de tipo $\textsc{Tprj}$ abaixo omite a condição de que o rótulo da operação de projeção deve ser um rótulo que de fato ocorre no registro. Uma expressão como $\sharp D\ \{A = 5, B = \texttt{true}, C = 80\}$, por exemplo, deve ser considerada mal-tipada pelo sistema de tipos. Essa verificação fica implícita na notação adotada na regra de tipo $\textsc{Tprj}$.

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n}{\Gamma \vdash \{lab_1 = e_1, \ldots lab_n = e_n\} : \{lab_1 : T_1, \ldots lab_n : T_n\}}\ (\textsc{Trcd})$$

$$\frac{\Gamma \vdash e : \{lab_1 : T_1, \ldots lab_n : T_n\}}{\Gamma \vdash \sharp lab_i\ e : T_i}\ (\textsc{Tprj})$$

Um par (e_1, e_2) pode ser visto como um açúcar sintático para registro $\{\texttt{fst} = e_1, \texttt{snd} = e_2\}$ com dois componentes e_1, e_2 identificados pelos rótulos de nome $\texttt{fst}$ e $\texttt{snd}$ respectivamente.

Observe que o programa abaixo, embora não leve a erro de execução, não é bem tipado pois a regra de tipo para aplicação (regra de tipo TAPP) requer que o tipo da entrada da função seja **igual** ao tipo do argumento

$$(\texttt{fn}\ x{:}\{B : \mathsf{int}, A : \mathsf{bool}\} \Rightarrow \texttt{if}\ \sharp A\ x\ \texttt{then}\ \sharp B\ x\ \texttt{else}\ 3)\ \ \{A = \texttt{true}, B = 10\}$$

No progrma acima o tipo da entrada da função, $\{A : \mathsf{bool},\ B : \mathsf{int}\}$, é (sintaticamente) diferente do tipo do argumento $\{B{:}\mathsf{int},\ A{:}\mathsf{bool}\}$.

Eis outro exemplo de programa que não leva a erro de execução, mas também não é considerado bem tipado:

$$(\texttt{fn}\ x{:}\{A : \mathsf{bool}\} \Rightarrow \texttt{if}\ \sharp A\ x\ \texttt{then}\ 2\ \texttt{else}\ 3)\ \ \{A = \texttt{true}, B = 10\}$$

Subtipos, a serem vistos mais tarde, flexibilizam o sistema de tipos e permitem a tipagem de casos como os acima

5.2 Listas e divisão

As extensões a serem vistas nessa subseção tornarão a linguagem L1 insegura, ou seja o sistema de tipos para L1 aumentado com regras para essas novas construções irá aceitar expressões que, quando avaliadas, levarão a *erros de execução*.

Exercício 33. *Defina as regras de tipo e da semântica operacional small step para expressões com a seguinte estrutura:* e_1/e_2

Exercício 34. *Explique por que o sistema de tipos de L1 com operação de divisão é inseguro em relação a semântica operacional*

Assim como pares ordenados e registros, listas são tipos de dados compostos, ou estruturados, ou ainda, tipos de dados não básicos. Registros são tipos de dados heterogêneos, pois seus componente podem ser de tipos diferentes. Já listas são **homogêneas** pois todos seus componentes devem ser do mesmo tipo.

Uma lista é ou é uma lista vazia, representada por `nil`, ou é uma lista não vazia construída através do operador binário "::" (que na sintaxe concreta em geral é infixado) que recebe como operandos um elemento da lista no lado esquerdo e no lado direito o restante da lista. Seguem abaixo alguns exemplos de listas em L1. No lado esquerdo do quadro abaixo as listas são escritas sem açúcar sintático, e no lado direito as mesmas listas são representadas usando um açúcar sintático:

Núcleo	Açúcar sintático
`nil`	[]
1 :: `nil`	[1]
8 :: (10 :: (5 :: `nil`))	[8,10,5]
(8 + 2) :: ((5 ∗ 10) :: (((fn x : int ⇒ x + 1) (10)) :: `nil`))	[8 + 2, 5 ∗ 10, (fn x : int ⇒ x + 1) 10]
(1,`true`) :: ((7, `false`) :: ((2, `true`) :: `nil`))	[(1,`true`), (7, `false`), (2, `true`)]

A primeira lista do quadro acima é a lista vazia. As duas listas seguintes são listas cujos componentes são todos números inteiros. A lista de quarta linha do quadro acima é uma lista de expressões que ao serem avaliadas vão produzir números inteiros. As listas das linhas 2, e e 4 do quadro acima são portanto do tipo `int list`. A última lista é uma lista de pares ordenados de inteiros com booleanos. O tipo dessa lista é portanto `(int * bool) list`.

Qual o tipo de `nil`? Como é a regra de tipo para a expressão `nil`? Observe que `nil` é uma lista, logo seu tipo será constuido com o construtor de tipo `list`. Mas lista de que tipo? Quando formos estudar tipagem implícita e polimorfismo vamos ter a seguinte regra de tipo para `nil` será $\Gamma \vdash nil : T\ list$ onde o tipo T

vai depender do contexto em que `nil` ocorre em uma expressão. Como aqui não estamos tipagem implícita e nem polimorfismo vamos começar com a seguinte versão de L1 com lista na qual o programador, toda vez que escrever a lista vazia em um programa L1, deve sempre anotar essa ocorrência com o seu tipo. Exemplos:

- `1 :: (5 :: nil:int))`
- `if e then nil:bool else true`

Além do operador binário :: para construção de listas, a linguagem L1 vem equipada com as seguintes operações unárias básicas sobre listas:

- operação `hd` que, dada uma lista não vazia retorna o seu primeiro elemento
- operação `tl` que, dada um lista não vazia, retorna a lista sem o seu primeiro elemento
- operação `isempty?` que, dada uma lista retorna `true` se a lista for vazia, e retorna `false` caso contrário

Exercício 35. *Escreva um programa em L1 que calcula a soma dos elementos de uma lista de inteiros e aplique essa função a lista* $[10, 30, 40, 20]$

Exercício 36. *Escreva um programa em L1 que define uma função* `map` *que recebe como argumento uma função de inteiros para inteiros e retorna como resultado um função de lista de inteiros para lista de inteiros. Seguem abaixo alguns exemplos que ilustram o comportamento de* `map`*:*

Uso de map	*resultado*
(`map` *(*`fn` x *: int* $\Rightarrow$ *x +1)) [10,20,30,50]*	*[11,21,31,51]*
(`map dobro`*) [2,4,8,10]*	*[4,8,16,20]*

Exercício 37. *Defina a extensão da sintaxe abstrata de L1 com listas e operações com listas.*

Exercício 38. *Defina a semântica operacional small step de L1 aumentada com listas e operações com listas.*

Exercício 39. *Defina a semântica operacional big step com substituições de L1 aumentada com listas e operações com listas.*

Exercício 40. *Defina a semântica operacional big step com ambiente de L1 aumentada com listas e operações com listas.*

Exercício 41. *Estenda a implementação em OCaml da inferência de tipos de L1 com listas e operações com listas. Para essa implementação considere que toda ocorrência de* `nil` *deve vir acompanhada do seu tipo.*

Exercício 42. *Estenda a implementação em OCaml de um dos avaliadores de L1 com listas e operações com listas.*

Exercício 43. *Defina o sistema de tipos de L1 aumentada com listas e operações com listas.*

Exercício 44. *O sistema de tipos definido no exercício 43 é seguro em relação a semântica operacional definida no exercício 38 ? Justifique a sua resposta.*

Exercício 45. *Defina (i) regras de tipo e (ii) regras da semântica operacional big step com ambientes para L1 aumentada com expressão de pattern matching* `match` e `with` $nil \Rightarrow e_1 \mid x::xs \Rightarrow e_2$. *Usando essa nova expressão (iii) escreva um programa em L1 que retorna o número de elementos de uma lista; (iv) escreva um programa em L1 que retorna a soma dos elementos de uma lista de números inteiros, e (v) um programa que mapeia uma função aos elementos de uma lista.*

5.3 Exceções

Há várias situações nas quais é necessário sinalizar que um evento excepcional ocorreu. Esse evento pode pode ser tratado pelo programa, ou pode fazer com que o programa encerre a sua execução. Algumas eventos que podem levar a uma exceção são: algum cálculo envolve divisão por zero ou *overflow*, a aplicação de *hd* ou *tl* a listas vazias, uma chave de busca ausente em um dicionário, índice de array está fora dos limites, um arquivo não foi encontrado ou não pode ser aberto, falta de memória suficiente, etc. Vamos aqui considerar três mecanismos simples de exceções para L1:

1. um evento excepcional simplesmente encerra a execução do programa (sem tratamento)
2. um evento excepcional transfere controle para um tratador de exceção
3. idem ao anterior mas passando informações adicionais para o tratador

5.3.1 Ativando exceções - sem tratamento

Vamos considerar uma extensão de L1 com a forma mais simples possível para sinalizar exceções: a inclusão de expressão `raise` a sintaxe da linguagem (algumas linguagens usam a palavra reservada *throw*)

$$e \quad ::= \quad \ldots \mid \texttt{raise}$$

Se o fluxo de execução atinge `raise` a exceção é propagada e o programa termina sua execução produzindo `raise` como resultado. É preciso adicionar uma série de regras na semântica operacional para propagar `raise`. Como exemplo, segue abaixo o conjunto de regras para avaliação do condicional. Além das 3 regras já existentes acrescentamos mais uma para propagar a exceção:

$$\texttt{if true then } e_2 \texttt{ else } e_3 \longrightarrow e_2 \quad (\textsc{if1}) \qquad \texttt{if false then } e_2 \texttt{ else } e_3 \longrightarrow e_3 \quad (\textsc{if2})$$

$$\texttt{if raise then } e_2 \texttt{ else } e_3 \longrightarrow \texttt{raise} \quad (\textsc{ifrs})$$

$$\frac{e_1 \longrightarrow e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3} \quad (\textsc{if3})$$

E abaixo seguem regras para aplicação com a adição de regras para propagar exceção. A regra APPERS descreve a situação em que o lado esquerdo de uma aplicação foi completamente avaliado e para `raise`. Já a regra APPELS descreve a situação em que o lado esquerdo terminou normalmente, mas o lado direito de uma aplicação resultou em `raise`. Em ambos os casos `raise` é propagado.

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \quad (\textsc{app1}) \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'} \quad (\textsc{app2})$$

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

$$\texttt{raise } e_2 \rightarrow \texttt{raise} \quad (\textsc{apprs}) \qquad v \texttt{ raise} \longrightarrow \texttt{raise} \quad (\textsc{appls})$$

A expressão raise já está completamente avaliada, mas ela **não** pode ser considerada como sendo um valor da linguagem. Caso considerássemos raise como sendo um valor, a semântica operacional deixaria de ser determinística

Note que temos a regra v raise $\rightarrow$ raise mas não temos regra e raise $\rightarrow$ raise. Isso porque a exceção no lado direito só será atingida quando o lado esquerdo da aplicação estiver completamente avaliado (lembre-se, a avaliação é da esquerda para direita). Caso a expressão e do lado direito entre em loop o raise do lado esquerdo não será "alcançado".

Vamos agora definir a regra de tipo para raise. Observe que raise pode ter qualquer tipo!!

- Em (fn x:bool $\Rightarrow$ x) raise, a expressão raise deverá ter tipo bool
- Em (fn x:bool $\Rightarrow$ x) (raise true), a expressão raise deverá ter tipo bool $\rightarrow$ bool

$$\Gamma \vdash \texttt{raise} : T \qquad (\textsc{Trs})$$

Um algoritmo *typeInfer* para a linguagem agora deverá ser mais "esperto". Ele deverá inferir o tipo de raise com base no seu contexto. Exemplo: a expressão $3 +$ raise é do tipo int e o contexto determina que o raise é do tipo int também.

5.3.2 Tratamento de exceções

A sintaxe de L1 é estendida com as seguintes expressões:

$$e \quad ::= \quad \ldots \mid \texttt{raise} \mid \texttt{try } e_1 \texttt{ with } e_2$$

A expressão try e_1 with e_2, quando avaliada, retorna o valor de e_1, mas se essa avaliação ativar uma exceção o tratador e_2 é avaliado. Todas as regras da semântica operacional da extensão de L1 com raise são mantidas e as seguintes regras são acrescentadas

$$\texttt{try } v_1 \texttt{ with } e_2 \rightarrow v_1 \qquad (\textsc{try}1) \qquad\qquad \texttt{try raise with } e_2 \rightarrow e_2 \qquad (\textsc{try}2)$$

$$\frac{e_1 \rightarrow e_1'}{\texttt{try } e_1 \texttt{ with } e_2 \rightarrow \texttt{try } e_1' \texttt{ with } e_2} \qquad (\textsc{try}3)$$

A expressão raise, como antes, continua tendo seu tipo definido pelo contexto. Note que a regra de tipo para try e_1 with e_2 requer que e_1 e e_2 tenham o mesmo tipo

$$\Gamma \vdash \texttt{raise} : T \qquad (\textsc{Trs}) \qquad\qquad \frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{try } e_1 \texttt{ with } e_2 : T} \qquad (\textsc{Try})$$

5.3.3 Tratamento de exceções com passagem de valores

No momento em que uma exceção é ativada, um valor pode ser passado como argumento para o seu tratador. Esse valor pode ser uma informação sobre a causa exata da exceção, ou pode ser um valor que ajude o tratador. Dessa forma, a expressão associada ao tratador deve ser uma função

$$\begin{array}{rcll} e & ::= & \ldots & \mid \texttt{raise}\ e \\ & & & \mid \texttt{try}\ e_1\ \texttt{with}\ e_2 \end{array}$$

Observe que agora uma expressão $\texttt{raise}\ v$ é uma formal normal. Ou seja já está completamente avaliada

$$\frac{e \longrightarrow e'}{\texttt{raise}\ e \longrightarrow \texttt{raise}\ e'} \quad \text{(E-Raise1)}$$

$$\texttt{raise}\ (\texttt{raise}\ v) \longrightarrow \texttt{raise}\ v \quad \text{(E-Raise2)}$$

$$\frac{e_1 \longrightarrow e_2}{\texttt{try}\ e_1\ \texttt{with}\ e_2 \longrightarrow \texttt{try}\ e_1'\ \texttt{with}\ e_2} \quad \text{(E-Try1)}$$

$$\texttt{try}\ v\ \texttt{with}\ e_2 \longrightarrow v \quad \text{(E-Try2)}$$

$$\texttt{try}\ (\texttt{raise}\ v)\ \texttt{with}\ e_2 \longrightarrow e_2\ v \quad \text{(E-Try3)}$$

Observe que as regras de tipo Trs-v e Try-v especificam que o valor a ser passado para o tratador de exceções na linguagem L1 estendida com exceções deve ser int. Esse valor inteiro pode ser interpretado como um código indicando a causa do erro.

$$\frac{\Gamma \vdash e_1 : \mathsf{int}}{\Gamma \vdash \texttt{raise}\ e_1 : T} \quad \text{(Trs-v)}$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : \mathsf{int} \to T}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : T} \quad \text{(Try-v)}$$

Um tratador tipicamente é organizado na forma de um *case* que identifica a causa da exceção e dá a ela um tratamento adequado.

Exercício 46. *Explique porque a linguagem deixaria de ser determinística caso* raise *fosse considerada um valor. Sugestão: pense como ficaria a avaliação da expressão* $(\texttt{fn}\ x{:}\mathit{int} \Rightarrow\ 0)\ \texttt{raise}$.

Exercício 47. *Note que em uma implementação o tipo de uma determinada ocorrência de* raise *deverá ser inferido de acordo com o contexto no qual ela aparece. Não seria mais simples exigir do programador que ele anote as ocorrências de* raise *com o tipo necessário? qual o problema com isso?*

Exercício 48. *Dê o tipo e o resultado da avaliação da seguinte expressão:*

$$(((\texttt{fn}\ x{:}\mathit{bool} \Rightarrow \texttt{fn}\ y{:}\mathit{bool} \Rightarrow \texttt{raise})\ \texttt{false})\ \texttt{false})\ \texttt{false}$$

Exercício 49. *Com o aumento da linguagem com exceções, o enunciado do teorema da progresso de tipos precisa ser modificado. Escreva esse enunciado.*

Exercício 50. *Usando as regras do sistema de tipos e da semântica operacional prove que o termo abaixo é bem tipado e o avalie*

$$
\begin{array}{l}
\texttt{try} \\
\qquad (\texttt{fn}\ x : \mathit{bool} \Rightarrow x)\ (\texttt{raise}\ 1) \\
\texttt{with} \\
\qquad \texttt{fn}\ z : \mathit{int} \Rightarrow \texttt{if}\ z = 0\ \texttt{then true else false}
\end{array}
$$

Capítulo 6

Segurança do sistema de tipos de L1

Vamos provar a segurança do sistema de tipos do núcleo de L1. A prova será por indução estrutural, usando a técnica de segurança sintática, que consiste em provar o Teorema do Progresso e da Preservação.

Teorema (Progresso). *Se* $\vdash e : T$ *então (i)* e *é valor ou (ii) existe* e' *tal que* $e \longrightarrow e'$.

Prova. Por indução na estrutura de e.

caso n Trivial pois n é valor.

casos `true`, `false`, $\texttt{fn}\ x : T \Rightarrow e$ Semelhantes ao caso anterior.

caso x Trivial pois $\vdash x : T$ é falso.

caso $\texttt{let rec}\ f : T_1 \to T_2 = (\texttt{fn}\ x : T_1 \Rightarrow e_1)\ \texttt{in}\ e_2$ Trivial pois pela regra E-LetRec temos que

$$\texttt{let rec}\ f : T_1 \to T_2 = (\texttt{fn}\ x : T_1 \Rightarrow e_1)\ \texttt{in}\ e_2 \longrightarrow \{\texttt{fn}\ x : T_1 \Rightarrow \texttt{let rec}\ f : T_1 \to T_2 = (\texttt{fn}\ x : T_1 \Rightarrow e_1)\ \texttt{in}\ e_1 / f\} e_2$$

caso $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2$ Assumimos

$$\vdash \texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 : T \tag{6.1}$$

e, como $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2$ não é valor, temos que provar:

$$\exists e'.\ \texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 \longrightarrow e' \tag{6.2}$$

Por T-Let com (6.1), temos $\vdash e_1 : T'$. Pela hipótese indutiva para e_1 temos que $\text{valor}(e_1) \vee \exists e'.\ e_1 \longrightarrow e'$.

Assumindo $\text{valor}(e_1)$: pela regra E-Let2 temos que $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 \longrightarrow \{e_1/x\} e_2$, provando (6.2).

Assumindo $\exists e'.\ e_1 \longrightarrow e'$: podemos afirmar que, para algum e_1', $e_1 \longrightarrow e_1'$, que usada como premissa para a regra E-Let1 nos permite concluir $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 \longrightarrow \texttt{let}\ x : T' = e_1'\ \texttt{in}\ e_2$, provando (6.2).

caso $\texttt{if}(e_1, e_2, e_3)$ Assumimos

$$\vdash \texttt{if}(e_1, e_2, e_3) : T \tag{6.3}$$

e, como $\texttt{if}(e_1, e_2, e_3)$ não é valor, temos que provar:

$$\exists e'.\ \texttt{if}(e_1, e_2, e_3) \longrightarrow e' \tag{6.4}$$

Por T-If com (6.3) temos $\vdash e_1 : \mathsf{bool}$. Pela hipótese indutiva para e_1 temos que $\mathsf{valor}(e_1) \ \vee \ \exists e'.\ e_1 \longrightarrow e'$.

Assumindo $\mathsf{valor}(e_1)$: como $\vdash e_1 : \mathsf{bool}$, pelas regras do sistema de tipos temos que $e_1 = \texttt{true}$ ou $e_1 = \texttt{false}$. Se $e_1 = \texttt{true}$, pela regra E-IfTrue temos que $\texttt{if}(e_1, e_2, e_3) \longrightarrow e_2$. Se $e_1 = \texttt{false}$, pela regra E-IfFalse temos que $\texttt{if}(e_1, e_2, e_3) \longrightarrow e_3$. Em ambos os casos, provamos (6.4).

Assumindo $\exists e'.\ e_1 \longrightarrow e'$: podemos afirmar que, para algum e_1', $e_1 \longrightarrow e_1'$, que, usada como premissa para a regra E-If nos permite concluir que $\texttt{if}(e_1, e_2, e_3) \longrightarrow \texttt{if}(e_1', e_2, e_3)$, provando (6.4).

caso $e_1 + e_2$ Assumimos

$$\vdash e_1 + e_2 : \mathsf{int} \tag{6.5}$$

e, como $e_1 + e_2$ não é valor, temos que provar:

$$\exists e'.\ e_1 + e_2 \longrightarrow e' \tag{6.6}$$

Pela regra T-Op+ e (6.5) temos que $\vdash \ e_1 : \mathsf{int}$. Pela hipótese indutiva temos que $\mathsf{valor}(e_1) \ \vee \ \exists e'.\ e_1 \longrightarrow e'$.

- Caso $\exists e'.\ e_1 \longrightarrow e'$: podemos afirmar que, para algum e_1', $e_1 \longrightarrow e_1'$, que usada como premissa para a regra E-Op1 nos permite concluir que $e_1 + e_2 \longrightarrow e_1' + e_2$, provando (6.6).
- Caso $\mathsf{valor}(e_1)$: pela regra T-Op+ e (6.5) temos que $\vdash \ e_2 : \mathsf{int}$. Pela hipótese indutiva temos que $\mathsf{valor}(e_2) \ \vee \ \exists e'.\ e_2 \longrightarrow e'$
  - Caso $\exists e'.\ e_2 \longrightarrow e'$: podemos afirmar que, para algum e_2', $e_2 \longrightarrow e_2'$, que usada como premissa para a regra E-Op2 nos permite concluir que $e_1 + e_2 \longrightarrow e_1 + e_2'$, provando (6.6).
  - Caso $\mathsf{valor}(e_2)$: como temos que $\vdash \ e_1 : \mathsf{int}$, e $\vdash \ e_2 : \mathsf{int}$, tanto e_1 como e_2 são números inteiros n_1 e n_2. Pela regra E-Op+ temos que $n_1 + n_2 \longrightarrow n$, provando (6.6).

caso $e_1 \ e_2$ Assumimos

$$\vdash e_1 \ e_2 : T \tag{6.7}$$

e, como $e_1 \ e_2$ não é valor, temos que provar:

$$\exists e'.\ e_1 \ e_2 \longrightarrow e' \tag{6.8}$$

Pela regra T-App e (6.7) temos que

$$\vdash \ e_1 : T' \to T \tag{6.9}$$

$$\vdash \ e_2 : T' \tag{6.10}$$

Por (6.9) e pela hipótese indutiva temos que $\mathsf{valor}(e_1) \ \vee \ \exists e'.\ e_1 \longrightarrow e'$.

- Se $\exists e'.\ e_1 \longrightarrow e'$: podemos afirmar que, para algum e_1', $e_1 \longrightarrow e_1'$, que usada como premissa para a regra E-App1 nos permite concluir que $e_1 \ e_2 \longrightarrow e_1' \ e_2$, provando (6.8).

- Se valor(e_1): pela hipótese indutiva com (6.10) temos que valor(e_2) $\vee$ $\exists e'.\ e_2 \longrightarrow e'$
  - se $\exists e'.\ e_2 \longrightarrow e'$: podemos afirmar que, para algum e_2', $e_2 \longrightarrow e_2'$, que usada como premissa para a regra E-App2 nos permite concluir que $e_1\ e_2 \longrightarrow e_1\ e_2'$, provando (6.8).
  - se valor(e_2): por (6.9) temos que e_1 é função anônima $\texttt{fn}\ x{:}T \Rightarrow e'$. Pela regra E-$\beta$ temos que $e_1\ e_2 \longrightarrow \{e_2/x\}e'$, provando (6.8). □

A prova do Teorema da Preservação usa o Lema da Substituição:

Lema (Substituição). *Se (i)* $\Gamma, x : T' \vdash e : T$ *e (ii)* $\Gamma \vdash e' : T'$ *então* $\Gamma \vdash \{e'/x\}\ e : T$.

Teorema (Preservação). *Se (i)* $\vdash e : T$ *e (ii)* $e \longrightarrow e'$ *então* $\vdash e' : T$.

Prova. Por indução na estrutura de e.

caso n Trivial pois n é valor, logo não há e' tal que $n \longrightarrow e'$.

casos `true`, `false`, $\texttt{fn}\ x : T \Rightarrow e$ Semelhantes ao caso anterior.

caso x Trivial pois $\vdash x : T$ é falso.

caso $\texttt{if}(e_1, e_2, e_3)$ Se $\vdash \texttt{if}(e_1, e_2, e_3) : T$ então pela regra de tipo TIf temos:

$$\vdash e_1 : \text{bool} \tag{6.11}$$

$$\vdash e_2 : T \tag{6.12}$$

$$\vdash e_3 : T \tag{6.13}$$

Se $\texttt{if}(e_1, e_2, e_3) \longrightarrow e'$ esse passo se deu por uma dentre essas 3 regra da semântica operacional*small step*: E-If, E-IfTrue, ou E-IfFalse. Vamos mostrar que, seja qual fora regra usada, o tipo é preservado:

Caso $\texttt{if}(e_1, e_2, e_3) \longrightarrow e'$ **por E-If**: nesse caso sabemos que

$$e_1 \longrightarrow e_1' \tag{6.14}$$

e que $e' = \texttt{if}(e_1', e_2, e_3)$. Temos que provar que $\vdash \texttt{if}(e_1', e_2, e_3) : T$. Por (6.11), (6.14), pela hipótese indutiva para e_1 e modus ponens temos que

$$\vdash e_1' : \text{bool} \tag{6.15}$$

Com (6.15), (6.12), e (6.13) como premissas para regra de tipo T-If concluimos $\vdash \texttt{if}(e_1', e_2, e_3) : T$.

Caso $\texttt{if}(e_1, e_2, e_3) \longrightarrow e'$ **por E-IfTrue**: nesse caso, $e_1 = \texttt{true}$ e $e' = e_2$. E $\vdash e_2 : T$ por (6.12).

Caso $\texttt{if}(e_1, e_2, e_3) \longrightarrow e'$ **por E-IfFalse**: nesse caso, $e_1 = \texttt{false}$ e $e' = e_3$. E $\vdash e_3 : T$ por (6.13).

caso $e_1 + e_2$ Se $\vdash e_1 + e_2 : T$ então pela regra de tipo T-Op+ temos que $T = \text{int}$ e

$$\vdash e_1 : \text{int} \tag{6.16}$$

$$\vdash e_2 : \text{int} \tag{6.17}$$

Se $e_1 + e_2 \longrightarrow e'$ esse passo se deu por uma dentre essas 3 regra da semântica operacional*small step*: E-Op1, E-Op2, ou E-Op+. Vamos mostrar que, seja qual fora regra usada, o tipo é preservado:

Caso $e_1 + e_2 \longrightarrow e'$ **por E-Op1**: nesse caso sabemos que

$$e_1 \longrightarrow e_1' \tag{6.18}$$

e que $e' = e_1' + e_2$. Temos que provar que $\vdash e_1' + e_2 : \text{int}$. Por (6.16), (6.18), pela hipótese indutiva para e_1 e modus ponens temos que

$$\vdash e_1' : \text{int} \tag{6.19}$$

Com (6.19) e (6.17) como premissas para regra de tipo T-OP+ concluímos $\vdash e_1' + e_2 : \text{int}$.

Caso $e_1 + e_2 \longrightarrow e'$ **por E-Op2**: nesse caso, e_1 é um valor, e sabemos que

$$e_2 \longrightarrow e_2' \tag{6.20}$$

e que $e' = e_1 + e_2'$. Temos que provar que $\vdash e_1 + e_2' : \text{int}$. Por (6.17), (6.20), pela hipótese indutiva para e_2 e modus ponens temos que

$$\vdash e_2' : \text{int} \tag{6.21}$$

Com (6.16) e (6.21) como premissas para regra de tipo T-OP+ concluimos $\vdash e_1 + e_2' : \text{int}$.

Caso $e_1 + e_2 \longrightarrow e'$ **por E-Op+**: nesse caso, $e_1 = n_1$, $e_2 = n_2$ e $e' = n$ (onde esse n é numeral inteiro que corresponde a soma dos inteiros n_1 e n_2. E temos que $\vdash n : \text{int}$.

caso $e_1\ e_2$ Se $\vdash e_1\ e_2 : T$ então pela regra de tipo T-App temos que e

$$\vdash e_1 : T' \to T \tag{6.22}$$

$$\vdash e_2 : T' \tag{6.23}$$

Se $e_1\ e_2 \longrightarrow e'$ esse passo se deu por uma dentre essas 3 regra da semântica operacional *small step*: E-App2, E-App2, ou E-β. Vamos mostrar que, seja qual for a regra usada, o tipo é preservado:

Caso $e_1\ e_2 \longrightarrow e'$ **por E-App1**: nesse caso sabemos que

$$e_1 \longrightarrow e_1' \tag{6.24}$$

e que $e' = e_1'\ e_2$. Temos que provar que $\vdash e_1'\ e_2 : T$. Por (6.22), (6.24), pela hipótese indutiva para e_1 e modus ponens temos que

$$\vdash e_1' : T' \to T \tag{6.25}$$

Com (6.25) e (6.23) como premissas para regra de tipo T-App concluímos $\vdash e_1'\ e_2 : T$.

Caso $e_1\ e_2 \longrightarrow e'$ **por E-App2**: nesse caso, e_1 é um valor, e sabemos que

$$e_2 \longrightarrow e_2' \tag{6.26}$$

e que $e' = e_1\ e_2'$. Temos que provar que $\vdash e_1\ e_2' : T$. Por (6.23), (6.26), pela hipótese indutiva para e_2 e modus ponens temos que

$$\vdash e_2' : T' \tag{6.27}$$

Com (6.22) e (6.27) como premissas para regra de tipo T-App concluímos $\vdash e_1\ e_2' : T$.

Caso $e_1\ e_2 \longrightarrow e'$ **por E-β**: nesse caso, $e_1 = fn\ x : T' \Rightarrow e$, a expressão e_2 é um valor v, e $e' = \{v/x\}\ e$. Temos que provar que $\vdash \{v/x\}\ e : T$. Como $\vdash fn\ x : T' \Rightarrow e : T' \to T$, pela regra de tipo T-Fn temos que

$$x : T' \vdash e : T \tag{6.28}$$

Pelo Lema da Substituição com (6.23) e (6.28), temos $\vdash \{v/x\}\ e : T$.

caso $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2$ Se $\vdash \texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 : T$ então pela regra de tipo T-Let temos que e

$$\vdash e_1 : T' \tag{6.29}$$

$$x : T' \vdash e_2 : T \tag{6.30}$$

Se $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 \longrightarrow e'$ esse passo se deu por uma dentre essas 2 regra da semântica operacional *small step*: E-Let1 ou E-Let2. Vamos mostrar que, seja qual for a regra usada, o tipo é preservado:

Caso $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 \longrightarrow e'$ **por E-Let1**: nesse caso sabemos que

$$e_1 \longrightarrow e_1' \tag{6.31}$$

e que $e' = \texttt{let}\ x : T' = e_1'\ \texttt{in}\ e_2$. Temos que provar que $\vdash \texttt{let}\ x : T' = e_1'\ \texttt{in}\ e_2 : T$. Por (6.29), (6.31), pela hipótese indutiva para e_1 e modus ponens temos que

$$\vdash e_1' : T' \tag{6.32}$$

Com (6.32) e (6.30) como premissas para regra de tipo T-Let1 concluímos $\vdash \texttt{let}\ x : T' = e_1'\ \texttt{in}\ e_2 : T$.

Caso $\texttt{let}\ x : T' = e_1\ \texttt{in}\ e_2 \longrightarrow e'$ **por E-Let2**: nesse caso, e_1 é um valor v, e $e' = \{v/x\}\ e_2$. Temos que provar que $\vdash \{v/x\}\ e_2 : T$. Pelo Lema da Substituição com (6.29) e (6.30), temos $\vdash \{v/x\}\ e_2 : T$.

caso $\texttt{let rec}\ f : T_1 \to T_2 = (\texttt{fn}\ x : T_1 \Rightarrow e_1)\ \texttt{in}\ e_2$ Com $\vdash \texttt{let rec}\ f : T_1 \to T_2 = (\texttt{fn}\ x : T_1 \Rightarrow e_1)\ \texttt{in}\ e_2 : T$, pela regra de tipo T-LetRec temos:

$$f : T_1 \to T_2, x : T_1 \vdash e_1 : T_2 \tag{6.33}$$

$$f : T_1 \to T_2 \vdash e_2 : T \tag{6.34}$$

Se $\texttt{let rec}\ f : T_1 \to T_2 = (\texttt{fn}\ x : T_1 \Rightarrow e_1)\ \texttt{in}\ e_2 \longrightarrow e'$, pela regra E-LetRec, $e' = \{\alpha/f\}\ e_2$ onde $\boldsymbol{\alpha} \equiv \texttt{fn}\ x{:}T_1 \Rightarrow \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ x{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_1$.

Se provarmos que

$$\vdash \alpha : T_1 \to T_2 \tag{6.35}$$

pelo Lema da Substituição com (6.34) e (6.35), provamos $\vdash \{\alpha/f\}\ e_2 : T$, como desejado.

Abaixo segue a prova de que $\vdash \alpha : T_1 \to T_2$:

Pelo Lema da da Permutação com (6.33) temos que

$$x : T_1, f : T_1 \to T_2 \vdash e_1 : T_2 \tag{6.36}$$

Observar que (6.36) é equivalente a

$$x : T_1, x : T_1, f : T_1 \to T_2 \vdash e_1 : T_2 \tag{6.37}$$

Com (6.37) e (6.36) construímos a derivação de tipo abaixo que prova $\vdash \alpha : T_1 \to T_2$:

$$\dfrac{\dfrac{x{:}T_1, f{:}T_1 \to T_2, x{:}T_1 \vdash e_1 : T_2 \qquad x{:}T_1, f{:}T_1 \to T_2 \vdash e_1 : T_2}{x{:}T_1 \vdash \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ x{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_1 : T_2}\ \textsc{Tletrec}}{\vdash \texttt{fn}\ x{:}T_1 \Rightarrow \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ x{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_1 : T_1 \to T_2}\ \textsc{Tfn}$$