2 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).

2.1 A Linguagem L1

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

Sintaxe de L1 (14)

$$
\begin{array}{rcl}
e & ::= & n \mid b \mid e_1 \ op\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
& \mid & x \\
& \mid & e_1\ e_2 \\
& \mid & \texttt{fn}\ x\texttt{:}T \Rightarrow e \\
& \mid & \texttt{let}\ x\texttt{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end} \\
& \mid & \texttt{let rec}\ f\texttt{:}T_1 \rightarrow T_2 = (\texttt{fn}\ y\texttt{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end}
\end{array}
$$

onde

$$
\begin{array}{rcl}
n & \in & \textit{conjunto de numerais inteiros} \\
b & \in & \{\texttt{true}, \texttt{false}\} \\
op & \in & \{+, \geq\}
\end{array}
$$

Na gramática acima:

- x representa um elemento pertencente ao conjunto $Ident$ de identificadores
- $\texttt{fn}\ x\texttt{:}T \Rightarrow e$ é uma função (sem nome)
- $e_1\ e_2$ é a aplicação da expressão e_1 a expressão e_2
- $\texttt{let}\ x\texttt{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end}$ é uma expressão que permite declarar identificadores
- $\texttt{let rec}\ f\texttt{:}T_1 \rightarrow T_2 = (\texttt{fn}\ y\texttt{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end}$ 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 $10 + \texttt{false}$ 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 $T_1 \to T_2 \to T_3$ é o mesmo que $T_1 \to (T_2 \to T_3)$
  - $\texttt{fn}$ se estende o mais a direita possível, logo $\texttt{fn}\ x{:}\mathsf{int} \Rightarrow x + x$ é o mesmo que $\texttt{fn}\ x{:}\mathsf{int} \Rightarrow (x + x)$

2.1.1 Semântica Operacional de L1

Vamos definir a semântica operacional de L1 no estilo conhecido por *semântica operacional estrutural* chamado também de *semântica operacional small-step*.

Relação de transição entre estados (16)

- Uma semântica operacional *small-step* é um *sistema de transição de estados*
- A relação de transição entre estados é chamada de $\longrightarrow$
- Escrevemos
$$c \quad \longrightarrow \quad c'$$
para dizer que há uma transição do estado c para o estado c'
- A relação $\longrightarrow^*$ é o fecho reflexivo e transitivo de $\longrightarrow$
- Escrevemos $c \not\longrightarrow$ quando não existe c' tal que $c \longrightarrow c'$
- um estado para L1 é uma expressão e

Valores e Erros de Execução (17)

- Valores são determinadas expressões (noção sintática)
- Os valores da linguagem L1 são dados pela seguinte gramática:
$$v ::= n \mid b \mid \texttt{fn}\ x{:}T \Rightarrow e$$

- Se $e \not\longrightarrow$ e a expressão e *não* é valor temos um *erro de execução* (noção semântica)
- Após vermos as regras da semântica operacional de L1 veremos que, por exemplo
  - `2+true` $\not\longrightarrow$

A relação de transição $\longrightarrow$ é definida através de um conjunto de regras de inferência da forma

$$\frac{premissa \ldots premissa}{conclusao}$$

Em uma semântica operacional *small step* não há regras para valores e, tipicamente, para cada construção da gramática abstrata que não é valor, temos:

- uma ou mais regras de *reescrita* (ou redução) e
- uma ou mais regras de *computação*

As regras de reescrita especificam a ordem na qual as subexpressões de uma expressão são avaliadas e as regras de computação dizem, de fato, como uma determinada expressão efetua uma computação interessante.

Semântica Operacional de Operações Básicas (18)

$$\frac{[\![n]\!] = [\![n_1]\!] + [\![n_2]\!]}{n_1 \texttt{+} n_2 \quad \longrightarrow \quad n} \qquad (\text{OP}+)$$

$$\frac{[\![b]\!] = [\![n_1]\!] \geq [\![n_2]\!]}{n_1 \geq n_2 \quad \longrightarrow \quad b} \qquad (\text{OP}\geq)$$

$$\frac{e_1 \quad \longrightarrow \quad e_1'}{e_1 \ op \ e_2 \quad \longrightarrow \quad e_1' \ op \ e_2} \qquad (\text{OP}1)$$

$$\frac{e_2 \quad \longrightarrow \quad e_2'}{v \ op \ e_2 \quad \longrightarrow \quad v \ op \ e_2'} \qquad (\text{OP}2)$$

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 (19)

$$\texttt{if true then } e_2 \texttt{ else } e_3 \quad \longrightarrow \quad e_2 \qquad (\text{IF}1)$$

$$\texttt{if false then } e_2 \texttt{ else } e_3 \quad \longrightarrow \quad e_3 \qquad (\text{IF}2)$$

$$\frac{e_1 \quad \longrightarrow \quad e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \quad \longrightarrow \quad \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3} \qquad (\text{IF}3)$$

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

Aplicação (20)

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

$$\frac{e_2 \quad \longrightarrow \quad e_2'}{v\ e_2 \quad \longrightarrow \quad v\ e_2'} \qquad (\text{APP}1)$$

$$\frac{e_1 \quad \longrightarrow \quad e_1'}{e_1\ e_2 \quad \longrightarrow \quad e_1'\ e_2} \qquad (\text{APP}2)$$

Substituição - Exemplos (21)

- 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:

$\{3/x\}(x+x) \equiv (3+3)$
$\{3/x\}((\texttt{fn}\ x:\mathsf{int} \Rightarrow x+y)\ x) \equiv (\texttt{fn}\ x:\mathsf{int} \Rightarrow x+y)\ 3$
$\{2/x\}(\texttt{fn}\ y:\mathsf{int} \Rightarrow x+y) \equiv \texttt{fn}\ y:\mathsf{int} \Rightarrow 2+y$

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 substituidas e que (ii) nenhuma variável livre ficará indevidamente ligada apos a substituição.
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 variávies devem ser declaradas.

$$
\begin{array}{lcl}
\{e/x\}\ x & = & e \\
\{e/x\}\ y & = & y\ \ (se\ x \neq y) \\
\{e/x\}\ \texttt{fn}\ y:T \Rightarrow e' & = & \texttt{fn}\ z:T \Rightarrow \{e/x\}\{z/y\}e' \\
 & & \text{se } x \neq y \ \text{ e } \ z \notin\ fv(e) \cup fv(e') \cup \{x,y\} \\
\{e/x\}\ \texttt{fn}\ x:T \Rightarrow e' & = & \texttt{fn}\ x:T \Rightarrow e' \\
\{e/x\}\ (e_1\ e_2) & = & (\{e/x\}e_1)\ (\{e/x\}e_2) \\
\{e/x\}\ n & = & n \\
\{e/x\}\ (e_1\ \mathrm{op}\ e_2) & = & \{e/x\}e1\ \mathrm{op}\ \{e/x\}e2 \\
\{e/x\}\ (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) & = & \texttt{if}\ \{e/x\}e1\ \texttt{then}\ \{e/x\}e2\ \texttt{else}\ \{e/x\}e3 \\
\{e/x\}\ b & = & b \\
\{e/x\}(\texttt{let}\ y{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end}) & = & \texttt{let}\ z{:}T = \{e/x\}e1\ \texttt{in}\ \{e/x\}\{z/y\}e_2\ \texttt{end} \\
 & & \text{se } x \neq y \ \text{ e } \ z \notin\ fv(e) \cup fv(e_2) \cup \{x,y\} \\
\{e/x\}(\texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end}) & = & \texttt{let}\ x{:}T = \{e/x\}e_1\ \texttt{in}\ e_2\ \texttt{end} \\
\{e/x\}(\texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end}) & = & \texttt{let rec}\ z:T_1 \to T_2 = \{e/x\}(\texttt{fn}\ y{:}T_1 \Rightarrow \{z/f\}e_1) \\
 & & \texttt{in}\ \{e/x\}\{z/f\}e_2 \\
 & & \texttt{end} \\
 & & \text{se } x \neq f \ \text{ e } \ z \notin\ fv(e) \cup fv(e_2) \cup \{x,f\} \\
\{e/f\}(\texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end}) & = & \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2\ \texttt{end}
\end{array}
$$

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

$$
\begin{array}{lcl}
fv(x) & = & \{x\} \\
fv(\texttt{fn}\ y:T \Rightarrow e) & = & fv(e) - \{y\} \\
fv(e_1 e_2) & = & fv(e_1) \cup fv(e_2) \\
fv(n) & = & \{\ \} \\
fv(e1\ \mathrm{op}\ e2) & = & fv(e1) \cup fv(e2) \\
\dots & \dots & \dots
\end{array}
$$

Como exercício termine a definição de fv iniciada acima.
A expressão let abaixo declara o identificador de nome f associado a uma função que soma um ao seu argumento. Esta função é aplicada a 10 na parte `in` da expressão:

$$\texttt{let}\ f{:}\mathsf{int} \to \mathsf{int} = \texttt{fn}\ x{:}\mathsf{int} \Rightarrow x+1\ \texttt{in}\quad f\ 10\ \texttt{end}$$

A expressão let pode ser considerada como simples açucar sintático:

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

Regras de redução (22)

$$\texttt{let}\ x{:}T = v\ \texttt{in}\ e_2\ \texttt{end} \quad \longrightarrow \quad \{v/x\}e_2 \qquad (\text{LET}1)$$

$$\frac{e_1 \quad \longrightarrow \quad e_1'}{\texttt{let}\ x{:}T = e_1\ \texttt{in}\ e_2\ \texttt{end} \quad \longrightarrow \quad \texttt{let}\ x{:}T = e_1'\ \texttt{in}\ e_2\ \texttt{end}} \qquad (\text{LET}2)$$

Funções recursivas (23)

A sintaxe

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

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 $(\texttt{fn}\ y{:}T_1 \Rightarrow e_1)$, ele substitui por $(\texttt{fn}\ y{:}T_1 \Rightarrow \texttt{let rec}\ f{:}T_1 \to T_2 = (\texttt{fn}\ y{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_1\ \texttt{end})$, que mantém o nome f preso em e_1 se alguma chamada recursiva a f precisar ser feita.

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

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
```

2.1.2 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 (24)

- o sistema de tipos de L1 consiste de um conjunto de regras de inferência

$$\frac{premissa \ldots premissa}{conclusao}$$

- Premissas e conclusão 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" onde:
  - Γ é um mapeamento finito de identificadores para seus tipos (ambiente de tipos). Denotamos por ε o ambiente vazio, e $(\Gamma, x : T)$ a extensão/redefinição de Γ fazendo a variável x apontar para o tipo T.
  - e é uma expressão da linguagem, e
  - T é um tipo pertencente ao conjunto definido pela seguinte gramática

$$T \quad ::= \quad \mathsf{int} \mid \mathsf{bool} \mid T \rightarrow T$$

    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 (25)

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

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

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

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \geq e_2 : \mathsf{bool}} \tag{T$\geq$}$$

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

Condicional (26)

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T} \tag{TIf}$$

- 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 $\texttt{if}\ 5 + 3 \geq 2\ \texttt{then true else}\ 5$, 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 pode concluir sobre o tipo de toda a expressão é preciso portanto, exigir que o tipo de ambas subexpressões seja o mesmo.

Tipando Funções (27)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \tag{TVar}$$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \texttt{fn}\ x : T \Rightarrow e : T \to T'} \tag{TFn}$$

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'} \tag{TApp}$$

Tipando expressão let (28)

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

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

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

2.1.3 Propriedades de L1

Propriedades (29)

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

Teorema 1 (Determinismo) Se $e \longrightarrow e'$ e se $e \longrightarrow e''$ 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 (30)

- 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 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*:

$$\textit{Se } \Gamma \vdash e : T \textit{ então } e \not\longrightarrow^* \textit{ erro}$$

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 (31)

- prova do **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 um passo pelas regras da semântica operacional. *Slogan*:

$$\textit{Se } \Gamma \vdash e : T \textit{ então } e \textit{ é valor, ou existe } e' \textit{ tal que } (e \quad \longrightarrow \quad e')$$

- prova da **preservação , após um passo da avaliação , do tipo de uma expressão**, ou seja, se uma expressão bem tipada progride em um passo, a expresssão resultante possui o mesmo tipo da expressão original. *Slogan*:

$$\textit{Se} \quad \Gamma \vdash e : T \quad e \quad (e \quad \longrightarrow \quad e') \quad \textit{então} \quad \Gamma \vdash e' : T.$$

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 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 (32)

Teorema 2 (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. ■

Teorema 3 (Preservação) Se $\vdash e : T$ e $(e \longrightarrow e')$ então $\vdash e' : T$.

Prova. Por indução na estrutura de e. ■

Note que, em ambos os casos, estamos assumindo que o ambiente de variáveis é vazio, isto é $\Gamma = \varepsilon$. Isto garante que as propriedades acima são válidas quando **não há variáveis livres** em e.

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

Problemas algorítmicos (33)

- 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 **Tipabilidade**: 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

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[1]. 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 (34)

Teorema 4 Dados ambiente Γ 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 Γ 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.

[1] fora alguns exercícios de verificação de tipos a serem feitos

2.2 A Linguagem L2

A linguagem L2 é uma extensão de L1 com memória (acesso e atualização de variáveis), sequência de comandos (';') e laço de repetição while. Primeiro, devemos estender a sintaxe de L1:

Sintaxe de L2 (35)

$$
\begin{array}{lrcl}
 & e & ::= & n \\
 & & | & b \\
 & & | & e_1 \; op \; e_2 \\
 & & | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
(*) & & | & l \; \texttt{:=} \; e \\
(*) & & | & \texttt{!} \; l \\
(*) & & | & \texttt{skip} \\
(*) & & | & e_1 \; \texttt{;} \; e_2 \\
(*) & & | & \texttt{while } e_1 \texttt{ do } e_2 \\
 & & | & \texttt{fn } x\texttt{:}T \Rightarrow e \\
 & & | & e_1 \; e_2 \\
 & & | & x \\
 & & | & \texttt{let } x\texttt{:}T = e_1 \texttt{ in } e_2 \texttt{ end} \\
 \\
 & v & ::= & n \\
 & & | & b \\
 & & | & \texttt{fn } x\texttt{:}T \Rightarrow e \\
(*) & & | & \texttt{skip}
\end{array}
$$

onde

- $n \in \mathbb{Z}$
- $b \in \{\mathsf{true}, \mathsf{false}\}$
- $op \in \{\geq, +\}$
- $l \in Loc = \{l_1, l_2, l_3, \ldots\}$

Intuição da sintaxe de L2 (36)

- *Loc* representa um conjunto de localizações de memória (ou endereços), cujos elementos vamos representar por $l, l_1, l_2, l_3, \ldots$. Assumimos que *Loc* é contável e infinito (existem tantas posições de memória quanto necessitarmos).
- Vamos assumir que somente *inteiros* serão armazenados em memória. Isso permite considerar a *memória* uma função total $\sigma : Loc \to \mathsf{int}$.
- l `:=` e representa o armazenamento do resultado da avaliação de e no endereço l dentro da memória.