Capítulo 6

A Linguagem L2

A linguagem L2 é uma extensão da linguagem L1 com construções imperativas para alocar, ler e escrever na memória. L2 também possui novas construções para controle do fluxo de execução (para sequência e repetição).

6.1 Sintaxe de L2

Programas em L2 pertencem ao conjunto definido pela gramática abstrata abaixo (as linhas marcadas com (*) indicam o que mudou em relação a L1 em termos de sintaxe abstrata):

Sintaxe de L2

$$
\begin{array}{lcl}
e & ::= & n \mid b \mid e_1 \ \mathit{op} \ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
(*) & \mid & e_1 \texttt{:=} e_2 \mid \texttt{!}\ e \mid \texttt{new}\ e \mid \boxed{\ell} \\
(*) & \mid & \texttt{skip} \mid e_1\texttt{;}\ e_2 \\
(*) & \mid & \texttt{while}\ e_1\ \texttt{do}\ e_2 \\
 & \mid & \texttt{fn}\ x\texttt{:}T \Rightarrow e \mid e_1\ e_2 \mid x \\
 & \mid & \texttt{let}\ x\texttt{:}T = e_1\ \texttt{in}\ e_2 \\
 & \mid & \texttt{let rec}\ f\texttt{:}T_1 \to T_2 = (\texttt{fn}\ y\texttt{:}T_1 \Rightarrow e_1)\ \texttt{in}\ e_2
\end{array}
$$

onde

$$
\begin{array}{lcl}
n & \in & \textit{conjunto de numerais inteiros} \\
b & \in & \{\texttt{true}, \texttt{false}\} \\
\mathit{op} & \in & \{+, \geq\} \\
\ell & \in & \textit{conjunto de endereços} \quad (Loc)
\end{array}
$$

As principais novidades em L2 são as seguintes:

- Programas em L2 podem operar com a memória
- Uma memória será abstraida como sendo uma função parcial de endereços para valores.
- A atribuição $e_1 := e_2$ requer que e_1 avalie para um local de memória e e_2 para um valor. Assim sendo, o seu efeito é armazenar o valor resultante de e_2 no local indicado por e_1.
- Os operadores unários new e ! são usados para alocar uma posição de memória e para acessar o valor contido em uma determinada posição de memória

- L2 possui o operador de composição sequencial de expressões ";" e também a expressão `while`, ambos comuns em linguagens imperativas

6.2 Semântica Operacional de L2

A semântica operacional no estilo *small step* consiste na definição da relação binária $\longrightarrow$ entre pares expressão, memória:

$$e, \sigma \longrightarrow e', \sigma'$$

Valores de L2

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

É comum, no estilo *small step* a necessidade de introduzir valores que surgem somente em passo intermediários da avaliação de um programa. **Esses valores intermediários não fazem parte da linguagem de programação disponível para o programador**. Na semântica operacional de L2 endereços, representados pelas metavariáveis l, l', etc, são valores intermediários.

Operações com Memória - Atribuição

$$\frac{l \in \mathit{Dom}(\sigma)}{\langle l := v,\ \sigma\rangle \longrightarrow \langle \texttt{skip},\ \sigma[l \mapsto v]\rangle} \quad (\text{ATR1})$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle v := e,\ \sigma\rangle \longrightarrow \langle v := e',\ \sigma'\rangle} \quad (\text{ATR2})$$

$$\frac{\langle e_1,\ \sigma\rangle \longrightarrow \langle e_1',\ \sigma'\rangle}{\langle e_1 := e_2,\ \sigma\rangle \longrightarrow \langle e_1' := e_2,\ \sigma'\rangle} \quad (\text{ATR3})$$

Observe que na linguagem L2, de acordo com as regras da semântica operacional, a avaliação de toda expressão L2

- ou termina com um valor,
- ou entra em loop,
- ou termina com "erro"

Se a avaliação de uma expressão $e_1 := e_2$ terminar com valor, esse valor será o `skip`.

Operações com Memória - alocação e derreferência

$$\frac{l \notin \mathit{Dom}(\sigma)}{\langle \mathsf{new}\ v,\ \sigma\rangle \longrightarrow \langle l,\ \sigma[l \mapsto v]\rangle} \quad (\textsc{ref}1)$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle \mathsf{new}\ e,\ \sigma\rangle \longrightarrow \langle \mathsf{ref}\ e',\ \sigma'\rangle} \quad (\textsc{ref}2)$$

$$\frac{l \in \mathit{Dom}(\sigma)\ \ \sigma(l) = v}{\langle !\ l,\ \sigma\rangle \longrightarrow \langle v,\ \sigma\rangle} \quad (\textsc{deref}1)$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma\rangle}{\langle !\ e,\ \sigma\rangle \longrightarrow \langle !\ e',\ \sigma\rangle} \quad (\textsc{deref}2)$$

Note, pelas regras da semântica, que a memória pode conter qualquer valor. Observe também que o endereço l criado por new e deve ser novo (na regra $\textsc{ref}1$ acima isso é especificado pela premissa $l \notin \mathit{Dom}(\sigma)$).

Além de construções que operam com a memória, a Linguagem L2 tem duas novas construçoes para controle de fluxo de execução: operador de execução sequencial, e `while`:

Controle de Fluxo - sequência e repetição

$$\langle \mathtt{skip};\ e_2,\ \sigma\rangle \longrightarrow \langle e_2,\ \sigma\rangle \quad (\textsc{seq}1)$$

$$\frac{\langle e_1,\ \sigma\rangle \longrightarrow \langle e_1',\ \sigma'\rangle}{\langle e_1;e_2,\ \sigma\rangle \longrightarrow \langle e_1';e_2,\ \sigma'\rangle} \quad (\textsc{seq}2)$$

$$\langle \mathtt{while}\ e_1\ \mathtt{do}\ e_2,\ \sigma\rangle \longrightarrow \langle \mathtt{if}\ e_1\ \mathtt{then}\ (e_2;\mathtt{while}\ e_1\ \mathtt{do}\ e_2)\ \mathtt{else}\ \mathtt{skip},\ \sigma\rangle \quad (\textsc{while})$$

Pela regra de reescrita $\textsc{seq}2$ acima, a avaliação sequencial de duas expressões é feita da esquerda para direita. Pela regra $\textsc{seq}1$ (uma regra de computação), quando o lado esquerdo estiver completamente reduzido para `skip`, a avaliação deve continuar com a expressão no lado direito do ponto e vírgula. Neste ponto o operador binário de execução sequencial ; difere dos demais operadores binários da linguagem.

Note que aqui foi feita uma escolha arbritrária no projeto da linguagem: a avaliação continua com a expressão no lado direito somente se a expressão do lado esquerdo do ponto e vírgula avalia para `skip`. Qualquer outra possibilidade leva a erro de execução.

A regra para `while` não se encaixa exatamente na classificação adotada até aqui para regras da semântica operacional *small step* pois ela pode ser compreendida como sendo tanto uma regra de reescrita como uma regra de computação.

Exercício 52. *Defina uma regra diferente de* $\textsc{Seq}1$ *de forma que o operador de avaliação sequencial seja*

tratado como um operador binário qualquer.

Exercício 53. *Defina uma semântica operacional no estilo big step para a linguagem L2.*

6.3 Sistema de Tipos para L2

Tipos para L2

- Como em $L2$ a memória pode conter valores de qualquer tipo, temos o tipo T ref que representa o tipo de endereço de memória que armazena um valor do tipo T.
- Note que endereços também podem ser armazenados, gerando tipos tais como (int ref) ref, por exemplo

$$\begin{array}{lll} T & ::= & \mathsf{int} \mid \mathsf{bool} \mid T_1 \to T_2 \\ (*) & \mid & T \text{ ref} \mid \mathsf{unit} \end{array}$$

Regras de Tipos para Operações com Memória

$$\frac{\Gamma \vdash e_1 : T \text{ ref} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \quad (\textsc{Tatr})$$

$$\frac{\Gamma \vdash e : T \text{ ref}}{\Gamma \vdash\ !\ e : T} \quad (\textsc{Tderef})$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \mathsf{new}\ e : T \text{ ref}} \quad (\textsc{Tref})$$

Observe que a regra de tipo para atribuição está de acordo com a regra de computação para a atribuição: uma atribuição, quando termina com valor produz skip e a regra de tipo associa o tipo unit para atribuições.

O tipo unit é bastante comum em muitas linguagens onde ele é conhecido como tipo void. O valor skip em algumas linguagens é conhecido como valor unit ou como valor ( ).

Skip e sequência

$$\Gamma \vdash \mathsf{skip} : \mathsf{unit} \quad (\textsc{Tskip})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{unit} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 ; e_2 : T} \quad (\textsc{Tseq})$$

O sistema de tipos atribui a skip o tipo unit. Sequências de ações $e_1 ; e_2$ são bem tipadas somente quando e_1 é do tipo unit. Neste caso, o tipo da sequência é igual ao tipo de e_2.

While

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \mathsf{unit}}{\Gamma \vdash \mathtt{while}\ e_1\ \mathtt{do}\ e_2 : \mathsf{unit}} \quad (\textsc{Twhile})$$

A regra de tipo de while e_1 do e_2 requer que e_1 seja uma expressão booleana, e que e_2 seja uma expressão do tipo unit (comando).

Observe que não há regra de tipos para endereços l, l', pois endereços não aparecem em programas fonte de L2.

Exercício 54. *Modifique a semântica operacional small step e o sistema de tipos de L2 de tal forma que uma expressão de atribuição $e_1 := e_2$, quando termina produzindo valor, esse valor seja o que resultar da avaliação da subexpressão e_2 e não necessariamente* skip.

Exercício 55. *Modifique a semântica operacional small step e o sistema de tipos de L2 de tal forma que, em uma expressão e_1; e_2, a subexpressão e_1 possa ser de qualquer tipo.*

Exercício 56. *Defina a semântica operacional big-step para L2.*

6.4 Exemplo de programas L2

O programa a seguir (com açúcar sintático) consiste de uma declaração da função fat em uma versão imperativa, seguida da aplicação fat 5.

```
let fat (x:int) : int =

        let z : int ref = new x
        let y : int ref = new 1

        while (!z > 0) (
           y :=  !y * !z;
           z :=  !z - 1;
        );
        ! y

fat 5
```

O mesmo programa após a remoção de açúcar sintático fica:

```
let fat : int-->int = fn x:int =>
        let z : int ref = new x in
        let y : int ref = new 1 in
          while (!z > 0) (
             y :=  (!y) * (!z);
             z :=  !z - 1;
          );
          ! y
in
    fat 5
```

O programa abaixo (com açúcar sintático) define um contador counter e uma função next_val que o incrementa toda vez que é chamada:

```
let counter : int ref = new 0

let next_val (z:unit) : int  =
      (
       counter := (!counter) + 1;
       !counter
      )

  (next_val skip)  +  (next_val skip)
```

O mesmo programa, agora na sua versão sem açúcar sintático:

```
let counter : int ref = new 0  in

let next_val : unit --> int  =
     fun (z:unit) =>
          (counter := (!counter) + 1;
          !counter)            in

(next_val skip)  +  (next_val skip)
```

6.5 Propriedades de L2

Como dito na seção anterior, programas na linguagem L2 não podem acessar diretamente endereços de memória, logo um sistema de tipos como uma especificação formal de um verificador de tipos para L2 **não precisa conter regra para tipar endereços** l uma vez que eles não aparecem em programas fonte mas somente em passos intermediários da avaliação.

Contudo, a técnica que adotamos para provar que o sistema de tipos é seguro em relação a semântica operacional envolve verificar se o tipo de expressões é preservado ao longo dos *small steps*, ou seja é preciso tipar expressões que surgem em etapas intermediárias da avaliação.

Dessa forma, **para fins de prova de segurança do sistema de tipos e somente para esse propósito**, é introduzida uma nova regra de tipo para endereços.

Em L2 localizações podem armazenar **qualquer valor**. Isso gera um problema para o sistema de tipos, pois uma expressão como $!l$ pode ter vários tipos distintos, dependendo do que estiver concretamente armazenado na memória na posição l. É necessário portanto introduzir um novo ambiente (de localizações) associando *localizações* a *tipos* T ref (o denotaremos por Δ).

Portanto, com a **única finalidade de provar segurança do sistema de tipos**, precisamos fazer as seguintes modificações nas regras de tipos:

- todas as regras já existentes se mantém exceto que todas as premissas e conclusões agora são da forma $\Gamma; \Delta \vdash e : T$
- é acrescentada a seguinte regra de tipo para endereços:

$$\frac{l \in \mathit{Dom}(\Delta) \qquad \Delta(l) = T \text{ ref}}{\Gamma; \Delta \vdash l : T \text{ ref}}$$

Perceba que, **para fazer a prova de segurança do sistema de tipos (e somente para isso)** todas as regras de tipo anteriores precisam ser modificadas para acomodar Δ.

Com memória capaz de armazenar qualquer valor fica um pouco mais trabalhosa a verificação da segurança do sistema de tipos de L2. A idéia central permanece a mesma, ou seja, é preciso provar que *expressões bem tipadas não levam a erro de execução quando avaliadas de acordo com a semântica operacional*. A primeira questão diz respeito à consistência entre Δ e σ, isto é, se o que o ambiente de tipos para localizações reflete o estado da memória.

Tipagem de memória

Definição: dizemos que uma memória σ está **bem tipada** em relação aos ambientes Γ e Δ, denotado por $\Gamma; \Delta \vdash \sigma$, quando

- $Dom(\Delta) = Dom(\sigma)$ e
- para todo $l \in Dom(\sigma)$, temos $\Delta(l) = T$ ref e $\Gamma; \Delta \vdash \sigma(l) : T$

Note também que um passo de avaliação $\langle e, \sigma \rangle \to \langle e', \sigma' \rangle$ permite alocar memória, o que torna possível que $Dom(\sigma')$ seja maior que $Dom(\sigma)$. Por conta disso, temos as seguintes definições para **preservação e progresso**.

Preservação

Se

- $\Gamma; \Delta \vdash e : T$,
- $\Gamma; \Delta \vdash \sigma$, e
- $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$

então existe $\Delta' \supseteq \Delta$ tal que

- $\Gamma; \Delta' \vdash \sigma'$.
- $\Gamma; \Delta' \vdash e' : T$

Progresso

Se

- $\Gamma; \Delta \vdash e : T$

então

- ou e é valor,
- ou, para toda memória σ tal que $\Gamma; \Delta \vdash \sigma$, $\langle e', \sigma' \rangle$ tal que $\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$.