L2 - extensão de L1 com construções imperativas

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

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

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

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

2 Semântica Operacional de L2

A semântica operacional *small step* consiste na definição da relação binária $\longrightarrow$ entre pares expressão, memória:

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

Valores de L2

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

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

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

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

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

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

Tipicamente em programas, o lado esquerdo de atribuições será uma variável, mas nada impede que um programador escreva um atribuições com uma expressão quaquer no lado esquerdo desde que do tipo adequado.

Operações com Memória - alocação e derreferência

$$\frac{l \notin \mathit{Dom}(\sigma)}{\langle \mathsf{new}\ v,\ \sigma\rangle \longrightarrow \langle l,\ \sigma[l \mapsto v]\rangle} \quad (\text{REF1})$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma'\rangle}{\langle \mathsf{new}\ e,\ \sigma\rangle \longrightarrow \langle \mathsf{ref}\ e',\ \sigma'\rangle} \quad (\text{REF2})$$

$$\frac{l \in \mathit{Dom}(\sigma)\ \ \sigma(l) = v}{\langle !\ l,\ \sigma\rangle \longrightarrow \langle v,\ \sigma\rangle} \quad (\text{DEREF1})$$

$$\frac{\langle e,\ \sigma\rangle \longrightarrow \langle e',\ \sigma\rangle}{\langle !\ e,\ \sigma\rangle \longrightarrow \langle !\ e',\ \sigma\rangle} \quad (\text{DEREF2})$$

Note que a memória pode conter qualquer valor. Observe também que o endereço l criado por $\mathsf{new}\ e$ deve ser novo, ou seja, um endereço ainda não alocado (na regra REF1 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

$$\langle () \ ;\ e_2,\ \sigma\rangle \longrightarrow \langle e_2,\ \sigma\rangle \quad (\text{SEQ1})$$

$$\frac{\langle e_1,\ \sigma\rangle \longrightarrow \langle e_1',\ \sigma'\rangle}{\langle e_1 ; e_2,\ \sigma\rangle \longrightarrow \langle e_1' ; e_2,\ \sigma'\rangle} \quad (\text{SEQ2})$$

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

Pela regra de reescrita SEQ2 acima, a avaliação sequencial de duas expressões é feita da esquerda para direita. Pela regra SEQ1 (uma regra de computação), quando o lado esquerdo estiver completamente reduzido

para (), a avaliação deve continuar com a expressão no lado direito do ponto e vírgula. Neste ponto a regra de semântica operacional small step do operador binário de execução sequencial ; difere da regra 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 (). 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 1. *Defina uma regra diferente de* SEQ1 *de forma que o operador de avaliação sequencial seja tratado como um operador binário qualquer.*

Exercício 2. *Defina uma semântica operacional no estilo big step para a linguagem L2.*

3 Sistema de Tipos para L2

Tipos para L2

- Como em $L2$ a memória pode conter valores de qualquer tipo, temos o tipo ref T que representa o tipo de endereço de memória que armazena um valor do tipo T.
- unit é o tipo de expressões que avaliam pelo seu efeito e não produzem nenhum valor interessante (na verdade produzem valor () - lido como valor "unit").

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

Note que endereços também podem ser armazenados, gerando tipos tais como ref (ref int), por exemplo.

Regras de Tipos para Operações com Memória

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

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

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

Observe que a regra de tipo para atribuição está de acordo com a sua regra de computação: uma atribuição, quando termina com valor produz () 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 () em algumas linguagens é conhecido como valor unit ou como valor ( ).

Skip e sequência

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

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

O sistema de tipos atribui a $()$ 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

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

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`.

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

Exercício 3. *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 $()$.*

Exercício 4. *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 5. *Defina a semântica operacional big-step para L2.*

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 :  ref int = new x
        let y :  ref int = 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 : ref int = new x in
        let y : ref int = 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 : ref int = new 0

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

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

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

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

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

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