Sintaxe de Tipos para a Linguagem L_2

Ver: `src/Types.ml`.

Existem, em `L2`, os seguintes tipos:

- `Int`, números inteiros
- `Bool`, booleanos
- `Unit`, unit (`()`, "void")
- `Reference t`, referência para um valor de tipo `t`

```
type tipo =
    | Int
    | Bool
    | Unit
    | Reference of tipo
    | ErrorType of string
;;
```

A fim de garantir a totalidade trivial (ou degenerada) do algoritmo de inferência estática de tipos, foi introduzido `ErrorType`, que representa um erro de tipagem obtido durante a inferência. Por exemplo,

```
if (1) then true else false : ErrorType "O tipo da condição `e1` deve ser um booleano, mas
foi `Int`"
```

Isso garante *apenas* que a inferência terminará. Não é uma garantia de que para todo termo tipável sobre L_2, o algoritmo `typeinfer e` será capaz de derivar o tipo de `e`.

`Int`, números inteiros

Esquema de Regra Abstrata

$$\frac{}{\Gamma \vdash e : \text{int}} \ (\text{T-Int})$$

Inferência de Tipo

```
    let rec typeinfer (e: term) (env: ambiente) : tipo = (match e with
        (** int n, número inteiro *)
        | Integer _ -> Int

        ...
    );;
```

isto é, qualquer **termo** inteiro (`Integer n`) tem o tipo `Int`. A implementação descarta o valor de `n`, pois não é importante para a tipagem; qualquer número inteiro é tipado como `Int`, etc.

Regra Concreta

```
let rec infer' (e: term) (env: ambiente) (r: type_inference) : (tipo * ambiente *
type_inference) = (match e with
        (** valores *)
        | Integer n -> (
            (Int, env, {
                name = "T-Int";
                pre = "T";
                post = string_of_env env ^ " ⊢ '" ^ ast_of_term e ^ "' : Int";
            } :: r)
        )

        ...)
;;
```

Pré-condição: `T` (sempre verdadeiro, sem pré-condição). Pós-condição: $\Gamma \vdash e : \text{int}$.

`Bool`, booleanos

Esquema de Regra Abstrata

$$\frac{}{\Gamma \vdash e : \text{bool}} \ (\text{T-Bool})$$

Inferência de Tipo

```
    let rec typeinfer (e: term) (env: ambiente) : tipo = (match e with
    ...
        (** b, booleano *)
        | Boolean _ -> Bool

        ...
    );;
```

isto é, qualquer **termo** booleano (`Boolean b`) tem o tipo `Bool`. A implementação descarta o valor de `b`, pois não é importante para a tipagem; qualquer booleano é tipado como `Bool`, etc.

Regra Concreta

```
let rec infer' (e: term) (env: ambiente) (r: type_inference) : (tipo * ambiente *
type_inference) = (match e with
    ...

        (** booleanos *)
        | Boolean b -> (
```

```
            (Bool, env, {
                name = "T-Bool";
                pre = "T";
                post = string_of_env env ^ " ⊢ '" ^ ast_of_term e ^ "' : Bool";
            } :: r)
        )

        ...
    );;
```

Pré-condição: `T` (sempre verdadeiro, sem pré-condição). Pós-condição: $\Gamma \vdash e : \text{bool}$.

`Unit`, unit (`()`, "void")

Esquema de Regra Abstrata

$$\frac{}{\Gamma \vdash e : \text{unit}} \ (\text{T-Unit})$$

Inferência de Tipo

```
    let rec typeinfer (e: term) (env: ambiente) : tipo = (match e with
    ...
        (** unit, unit *)
        | Unit -> Unit

        ...
    );;
```

Regra Concreta

```
let rec infer' (e: term) (env: ambiente) (r: type_inference) : (tipo * ambiente *
type_inference) = (match e with
    ...

        (** unit, unit *)
        | Unit -> (
            (Unit, env, {
                name = "T-Unit";
                pre = "T";
                post = string_of_env env ^ " ⊢ '" ^ ast_of_term e ^ "' : Unit";
            } :: r)
        )

        ...
    );;
```

Pré-condição: `T` (sempre verdadeiro, sem pré-condição). Pós-condição: $\Gamma \vdash e : \text{unit}$.

`Reference t`, referência para um valor de tipo `t`

Este é o único tipo não-trivial em L_2. Uma referência a um tipo `t`, ou `Reference t` é o tipo *apontador para tipo* `t`.

Esquema de Regra Abstrata

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{ref } e : \text{ref } t} \text{ (T-Ref)}$$

Inferência de Tipo

```
    let rec typeinfer (e: term) (env: ambiente) : tipo = (match e with
    ...
        (** ref e, aloca uma referência para um valor de tipo t *)
        | New e1 ->
            let t1 = typeinfer e1 env in
            (match t1 with
            | ErrorType msg -> ErrorType msg
            | _ -> Reference t1)

        ...
    );;
```

Explicação:

1. Primeiro inferimos t_1 = typeinfer e1 env
    1. Se t_1 for um erro, propagamos o erro
    2. Caso contrário, devolvemos Ref t_1

Assim, o tipo de `New e1` é sempre `Reference (tipo(e1))`, conforme a regra abstrata.

Regra Concreta

```
let rec infer' (e: term) (env: ambiente) (r: type_inference)
    : (tipo * ambiente * type_inference) = (match e with
    ...

        (** Reference  e, aloca um valor e cria uma referência *)
        | New e1 ->
            let (t1, env1, r1) = infer' e1 env r in
            let t =
                match t1 with
                | ErrorType _ -> t1
                | _ -> Reference t1
```

```
            in
            (t, env1, {
                name = "T-New";
                pre =
                    (match t1 with
                     | ErrorType _ -> "T (erro propagado)"
                     | _ -> string_of_env env ^
                            " ⊢ '" ^ ast_of_term e1 ^ "' : " ^ string_of_type t1);
                post =
                    (match t with
                     | ErrorType _ -> string_of_type t
                     | _ -> string_of_env env ^
                            " ⊢ ref(" ^ ast_of_term e1 ^ ") : " ^ string_of_type t);
            } :: r1)

        ...
    );;
```

Note que a **regra que cria novas referências** é `T-New`, isto é, a regra de tipagem para o termo `new e`. Como veremos em `termos/new e`, `new e : Reference (typeinfer e)`.

Pré-condição: $\Gamma \vdash e : t$. Pós-condição: $\Gamma \vdash \text{new } e : \text{Reference } t$.

Inferência Estática de Tipos

1. A inferência de tipos é feita pela função `infer e` em `src/TypeInference.ml`.

É uma função que chama a função recursiva `let rec infer' (e: term) (env: ambiente) (r: type_inference) : (tipo * ambiente * type_inference)`.

Note que, cada vez que `infer'` é chamada, é necessário fornecerum term e um ambiente de tipos. Também é passada a lista de regras de inferência obtidas até este ponto. As regras vão sendo acumuladas em uma lista, e ao final da inferência esta lista de regras é devolvida ao programador.

2. Uma regra de inferência é um objeto do tipo

```
(** `src/Constructions.ml` *)

type rule = {
    name:   string;
    pre:    string;
    post:   string;
}   and type_inference = rule list
    and evaluation = rule list;;
```

Note que uma inferência de tipo (`type_inference`) é uma lista de regras de inferência de tipos, e uma avaliação de termos (`evaluation`) é uma lista de regras de avaliação de termos.

3. O tipo de `infer` é `infer (e: term) : tipo * type_inference`. Dado um termo `e`, `infer` retorna o tipo de `e` e a lista de regras de inferência usadas para tipá-lo.

4. O ambiente de tipos `env` é passado em todos os passos de `infer'`. O ambiente de tipos é somente uma lista de pares (identificador: string * tipo: tipo). As funções `lookup` e `put` usadas para lidar com o ambiente de tipos são definidas em `src/Constructions.ml`.

Outras Coisas

1. `src/Representations.ml`
   Representações em formato de string para termos, valores, tipos, ambiente de tipos, memória, regras de inferência de tipos e avaliação de termos. É comum que funções que retornam representação em string de um objeto, em OCaml, sejam escritas `string_of_t`, para algum tipo `t`, então `string_of_term` é uma função que recebe um `term` e retorna sua representação como `string`;
2. `src/Constructions.ml`
   Este módulo define todas as *construções* sobre termos ou sobre tipos:

ambiente de tipos: `ambiente`;

Um `ambiente` é uma lista de `name_binding`. Um `name_binding` é um par `string` (identificador, `x`) e um `tipo`. Com isso, **associamos** um **nome** a um **tipo**. Um **ambiente de tipos**, então´, é uma **lista de associações entre nomes e tipos**, ou

```
type ambiente = (string * tipo) list
```

Existem duas funções sobre ambientes de tipos: `lookup x env` e `put x t env`.

`lookup x env` retorna o tipo de `x` em `env`, se `x` estiver em `env`, e `None` caso contrário. `put x t env` adiciona/atualiza (x,t) no ambiente.

`put x t env` coloca o par `x` e `t` no ambiente `env` e retorna o ambiente resultante.

regras de inferência de tipos e avaliação de termos

```
(**
    Esquema de regra concreto para inferência estática de tipos
    e para avaliação de termos
**)
type rule = {
    name:   string;
    pre:    string;
    post:   string;
}   and type_inference = rule list
    and evaluation = rule list;;
```

note que uma inferência de tipo (`type_inference`) é uma lista de regras [de inferência de tipo], e uma avaliação de termo (`evaluation`) é uma lista de regras [de avaliação de termos]. Etc.

tabela de símbolos

```
(** localização na memória, aliás int *)
type location = int;;
```

```
(** uma tabela de símbolos é uma lista de pares (identificador, localização)
    e representa um mapeamento entre variáveis e posições na memória *)
type symbols = (string * location) list;;
```

em [nossa implementação de] L_2, a memória é feita em dois passos:

1. identificadores são armazenados em uma *tabela de símbolos*, que associa identificadores a localizações na memória;
2. a memória é uma associação entre localizações e valores.

tanto a tabela de símbolos quanto a memória são listas: `symbols` é uma lista de pares `string * location` e `memory` é uma lista de pares `location * value`.

métodos sobre `symbols`:

```
(** verifica se a variável `x` está na tabela de símbolos `table` *)
let rec is_bound (x: string) (table: symbols) : bool = match table with
    | [] -> false
    | (y, l) :: table -> if x = y then true else is_bound x table
;;

(** retorna a localização da variável `x` na tabela de símbolos `table` *)
let rec search (x: string) (table: symbols) : location option = match table with
    | [] -> None
    | (y, l) :: table -> if x = y then Some l else search x table
;;

(** adiciona uma nova variável `x` na tabela de símbolos `table` *)
let rec extend (x: string) (l: location) (table: symbols) : symbols = (x, l) :: table;;
```

memória

```
(**
    uma memória é uma lista de pares (localização, valor) e representa um mapeamento entre
localizações e valores *)
type memory = (location * value) list;;

let rec exists (loc: location) (mem: memory) : bool = match mem with
    | [] -> false
    | (l, v) :: mem -> if loc = l then true else exists loc mem
;;

let rec get (loc: location) (mem: memory) : value option = match mem with
    | [] -> None
    | (l, v) :: mem -> if loc = l then Some v else get loc mem
;;

(** atualiza um local na memória e reordena a memória de acordo com a localização (fst mem)
*)
let rec set (loc: location) (v: value) (mem: memory) : memory = (
    let rec aux acc = function
        | [] -> List.rev ((loc, v) :: acc)
```

```
        | (l, vv) :: tail ->
            if l = loc then List.rev_append acc ((loc, v) :: tail)
            else aux ((l, vv) :: acc) tail
    in
    aux [] mem
)

(** ordena a memória em função das localizações (loc, fst mem) *)
let sort (mem: memory) : memory =
    let compare (l1, _) (l2, _) = compare l1 l2 in
    List.sort compare mem
;;

(** retorna a próxima posição livre na memória *)
let where mem =
    mem
    |> List.map fst
    |> List.sort compare
    |> List.fold_left (fun expect l ->
            if l = expect then expect + 1 else expect
        ) 0
```

Termos

Sintaxe de termos

Sejam Γ o ambiente de tipos e seja σ a memória e $\mathrm{símbolos}$ a tabela de símbolos.

```
(** sintaxe de termos sobre `L2` *)
type term =
    | Integer of int                                    (* n, termo número inteiro *)
    | Boolean of bool                                   (* b, termo booleano *)
    | Identifier of string                              (* x, identificador *)
    | Conditional of term * term * term                 (* If, operador condicional *)
    | BinaryOperation of binary_operator * term * term    (* bop, operação binária *)
    | While of term * term                              (* While e1 do e2 *)
    | Assignment of term * term                         (* x := e *)
    | Let of string * tipo * term * term                (* let x: T = e1 in e2 *)
    | New of term                                       (* new e *)
    | Derefence of term                                 (* !e *)
    | Unit                                              (* () *)
    | Sequence of term * term                           (* e1; e2 *)
    | Location of int                                   (* l, local de memória *)
and binary_operator =
      | Add | Sub | Mul | Div                           (* operadores aritméticos *)
      | Eq  | Neq | Lt  | Leq | Gt | Geq                (* operadores relacionais *)
      | And | Or                                        (* operadores lógicos *)
;;
```

Valores

Ver `src/Terms.ml`

Não existem propriamente, em L_2, *regras de avaliação* para *valores*, uma vez que valores são termos já avaliados; são formais normais ou presas, *stuck*.

São valores os termos

```
type term = ..
    | Integer of int        (*  termo número inteiro *)
    | Boolean of bool       (*  termo booleano *)
    | Unit                  (*  termo unitário *)
    | Location of int       (*  local de memória *)
```

Estes termos são imediatamente avaliados para seus respectivos valores.

```
(** sintaxe de valores sobre `L2` *)
type value =
    | VInteger of int                               (* n, valor número inteiro *)
    | VBoolean of bool                              (* b, valor booleano *)
    | VUnit                                         (* (), unit *)
    | VLocation of int                              (* l, local de memória *)
    | EvaluationError of string                     (* s, erro de avaliação *)
and name_binding = string * value                   (* associação entre um identificador e
um valor *)
;;
```

O termo `Location` e o valor `VLocation` são discutidos `docs/outras_coisas.md` #memória.

Note que `EvaluationError` não é um termo da linguagem L_2. Ao contrário, `EvaluationError` é definido somente na sintaxe dos *valores*. `EvaluationError` não é um valor da linguagem `L_2`, também, mas é o valor resultante da avaliação de um termo mal-formado ou que a avaliação falhou.

Funções *helper*

1. `is_value_term` : $\text{term} \to \mathbb{B}$, verifica se um termo representa um valor.
2. `value_of_term` : $\text{term} \to \text{value} \to \mathbb{B}$, converte um termo para um valor, (caso `is_value_term e` retorne verdadeiro para `e`).
3. `term_of_value` : $\text{value} \to \text{term} \to \mathbb{B}$, converte um valor para um termo, (caso `value_of_term e` retorne verdadeiro para `e`).
4. `substitute` $\text{string} \times \text{value} \times \text{term} \to \text{term}$, substitui um identificador por um valor em um termo. {x/v}e.

Regras de Avaliação

Artificialmente, definimos as regras de avaliação para valores, a fim de, ao imprimí-las em tela, sabermos em que ponto da avaliação foi produzido o valor.

$$\frac{}{\text{Integer n},\ \sigma\ \to \text{VInteger n},\ \sigma}\ (\text{E-Int})$$

$$\frac{}{\text{Boolean b},\ \sigma\ \to \text{VBoolean b},\ \sigma}\ (\text{E-Bool})$$

$$\frac{}{\text{Unit},\ \sigma\ \to \text{VUnit},\ \sigma}\ (\text{E-Unit})$$

$$\frac{}{\text{Location l, } \sigma \ \rightarrow \text{VLocation l, } \sigma} \ (\text{E-Location})$$

```
(** faz um passo na avaliação de um termo, se for possível *)
let rec step    (e     :    term)
                (mem   :  memory)
                : (term * memory * eval_rule, string) result =
    match e with
    ...
    |   Integer n -> Ok (VInteger n, mem, {
                            name = "E-Int";
                            pre  = "";
                            post =  ast_of_term e ^ ", " ^ string_of_mem mem ^ " -> "
                            ^ string_of_value (VInteger n) ^ ", " ^ string_of_mem mem
                        })

    |   Boolean b -> Ok (VBoolean b, mem, {
                            name = "E-Bool";
                            pre  = "";
                            post = ast_of_term e ^ ", " ^ string_of_mem mem ^ " -> "
                            ^ string_of_value (VBoolean b) ^ ", " ^ string_of_mem mem
    })

    |   Unit      -> Ok (VUnit, mem, {
                            name = "E-Unit";
                            pre  = "";
                            post = ast_of_term e ^ ", " ^ string_of_mem mem ^ " -> "
                            ^ string_of_value (VUnit) ^ ", " ^ string_of_mem mem
    })

    |   Location l -> Ok (VLocation l, mem, {
                            name = "E-Location";
                            pre  = "";
                            post = ast_of_term e ^ ", " ^ string_of_mem mem ^ " -> "
                            ^ string_of_value (VLocation l) ^ ", " ^ string_of_mem mem
    })

    ...
```

Identificadores (variáveis, `Identifier x`)

Inferência de Tipo

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \ (\text{T-Var})$$

Se o identificador `x` estiver no ambiente de tipos `Γ` com o tipo `T`, então o tipo do identificador `x` é `T`.

Avaliação

L_2 não define explicitamente a avaliação de identificadores.

Identificadores são as variáveis que o programador define em seus programas. Estas variáveis são mapeadas em uma *tabela de símbolos*, que é uma lista de pares (`identificador` : string × `localização na memória` : int

). Um identificador **não** é o valor que ele aponta. Ao contrário, um identificador **avalia para** a **posição na memória** *apontada* por ele.

Se `x` estiver na *tabela de símbolos*, então `Identifier x` avalia para o valor posição *na memória* `VLocation l` para o qual `x` aponta. Se `x` não estiver na *tabela de símbolos*, então `Identifier x` avalia para `EvaluationError`.

$$\frac{\text{x} \in \text{símbolos},\ \sigma}{\sigma \vdash \text{Identifier x} \to \text{VLocation } l} \quad \text{(E-Var)}$$

Condicional (`if e1 then e2 else e3`)

Sintaxe: `Conditional (e1, e2, e3)`

Inferência de Tipo

A expressão condicional exige que:

- a condição e_1 tenha tipo `Bool`;
- ambos os ramos e_2 e e_3 tenham o $\mathbf{mesmo\ tipo}$ T.

A regra formal é:

$$\frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \quad \text{(T-If)}$$

Se e_1 não for booleano, ou se e_2 e e_3 tiverem tipos distintos, então a expressão não tem tipo válido.

Avaliação

- Primeiro reduzimos a condição e_1, caso ela ainda não seja um valor.
- Se e_1 reduz para `true`, escolhemos o ramo e_2.
- Se e_1 reduz para `false`, escolhemos o ramo e_3.

As regras formais são as seguintes:

$$\frac{e_1 \to e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \ \to\ \text{if } e_1' \text{ then } e_2 \text{ else } e_3} \quad \text{(E-IfStep)}$$

$$\frac{}{\text{if true then } e_2 \text{ else } e_3 \ \to\ e_2} \quad \text{(E-IfTrue)}$$

$$\frac{}{\text{if false then } e_2 \text{ else } e_3 \ \to\ e_3} \quad \text{(E-IfFalse)}$$

O condicional avalia primeiro a condição.
Se ela for verdadeira, retorna o ramo `then`.
Se for falsa, retorna o ramo `else`.
Se ainda não for um valor booleano, a condição é reduzida antes de continuar.

Operações Binárias (`BinaryOperation(op, e1, e2)`)

Sintaxe: `BinaryOperation (op, e1, e2)`

Inferência de Tipo

A regra de tipos para operações binárias depende do operador `op`.
Dividimos em três classes:

1. **Operadores aritméticos:** `+`, `-`, `*`, `/`, `mod`
   - Ambos os operandos devem ser do tipo `Int`
   - O resultado é `Int`

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 \ \text{op} \ e_2 : \text{Int}} \quad (\text{T-BinOp } \{+, \text{-}, *, /, \text{mod}\})$$

2. **Operadores booleanos:** `and`, `or`
   - Ambos os operandos devem ser `Bool`
   - O resultado é `Bool`

$$\frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \text{Bool}}{\Gamma \vdash e_1 \ \text{op} \ e_2 : \text{Bool}} \quad (\text{T-BinOp } \{\text{and}, \text{or}\})$$

3. **Operadores relacionais:** `=`, `<>`, `<`, `<=`, `>`, `>=`
   - Ambos os operandos devem ser `Int`
   - O resultado é `Bool`

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash e_1 \ \text{op} \ e_2 : \text{Bool}} \quad (\text{T-BinOp } \{=, <>, <, <=, >, >=\})$$

Também é possível comparar booleanos com `==` e `<>`.

Se os tipos não coincidirem com o esperado pelo operador, a expressão binária não tem tipo válido.

Avaliação

A avaliação segue a ordem esquerda–direita:

1. Reduzimos `e1` até virar um valor.
2. Depois reduzimos `e2`.
3. Quando ambos são valores, aplicamos o operador.

As regras formais são:

$$\frac{e_1 \ \to \ e_1'}{\text{op}(e_1, e_2) \ \to \ \text{op}(e_1', e_2)} \quad (\text{E-BinOp 1})$$

$$\frac{e_2 \to e_2'}{\text{op}(v_1, e_2) \ \to \ \text{op}(v_1, e_2')} \quad (\text{E-BinOp 2})$$

E quando ambos os operandos são valores:

$$\frac{}{\text{op}(v_1, v_2) \ \to \ v} \quad (\text{E-BinOp})$$

onde `v` é o resultado da computação real do operador sobre os valores v_1 e v_2, por exemplo:

- `1 + 2 → 3`

- `4 < 2 → false`
- `true and false → false`

Ordem de avaliação: operações binárias avaliam sempre primeiro o lado esquerdo, depois o direito, e finalmente aplicam o operador quando ambos os operandos forem valores.

Comando de Atribuição (`Assignment(x, e)`)

Sintaxe: `Assignment (Identifier x, e)`
Semântica: **modifica a memória**, armazenando o valor de `e` na posição referenciada por `x`.

Inferência de Tipo

Para que a atribuição seja válida:

1. `x` deve ter tipo `ref T` no ambiente.
2. `e` deve ter tipo `T`.
3. O comando retorna tipo `Unit`.

A regra formal é:

$$\frac{\Gamma \vdash x : \text{Ref } T \qquad \Gamma \vdash e : T}{\Gamma \vdash x := e : \text{Unit}} \quad (\text{T-Atr})$$

Se `x` não for uma referência, ou `e` não tiver o tipo apontado por `x`, a atribuição é inválida.

Avaliação

A avaliação segue a ordem:

1. Avaliamos `x` **até obter a localização** `l`.
2. Avaliamos `e` até obter o valor `v`.
3. Escrevemos `v` na memória no endereço `l`.
4. O comando retorna `Unit`.

Regras formais:

Avaliar o lado esquerdo:

$$\frac{x \to x'}{x := e \ \to \ x' := e} \quad (\text{E-Atr})$$

Avaliar o lado direito:

$$\frac{e \to e'}{l := e \ \to \ l := e'} \quad (\text{E-Atr})$$

Quando ambos lados são valores:

$$\frac{}{l := v \ \to \ \text{store}(l \mapsto v)\ \text{VUnit}} \quad (\text{E-Atr})$$

A atribuição **muda a memória** e é avalia para **valor** `VUnit`.

Laço `while` (`While(e1, e2)`)

Sintaxe: `While (e1, e2)`
Semântica: Enquanto a condição `e1` for verdadeira, executar o corpo `e2`.

Inferência de Tipo

Para um `while` ser bem tipado:

- a condição `e1` deve ser `Bool`;
- o corpo `e2` deve ter tipo `Unit`;
- o tipo da expressão inteira é `Unit`.

$$\frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \text{Unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{Unit}} \quad \text{(T-While)}$$

Avaliação

Um `while` é açúcar sintático para:

$$\text{while } e_1 \text{ do } e_2 \quad \equiv \quad \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else Unit}$$

A avaliação segue a mesma estratégia:

1. Avaliamos `e1`.
2. Se `e1 → true`, então avaliamos `e2` e depois repetimos o loop.
3. Se `e1 → false`, o laço termina retornando `Unit`.

Regras formais:

Passo sobre a condição:

$$\frac{e_1 \to e_1'}{\text{while } e_1 \text{ do } e_2 \ \to \ \text{while } e_1' \text{ do } e_2} \quad \text{(E-While Step)}$$

Caso verdadeiro:

$$\frac{}{\text{while true do } e_2 \ \to \ e_2; \ \text{while true do } e_2} \quad \text{(E-While True)}$$

Caso falso:

$$\frac{}{\text{while false do } e_2 \ \to \ \text{Unit}} \quad \text{(E-While False)}$$

Assim como em linguagens imperativas reais, o laço `while` produz sempre `Unit` — ele serve apenas para efeitos colaterais na memória.

Sequência de Comandos (`e1 ; e2`)

Sintaxe: `Sequence (e1, e2)`
Semântica: **Avalia** `e1`, ignora seu valor (que deve ser `Unit`), e depois avalia `e2`, cujo valor é o resultado da sequência.

Inferência de Tipo

Para uma sequência ser bem tipada:

1. `e1` deve ter tipo `Unit`.
2. `e2` pode ter qualquer tipo `T`.
3. O tipo da sequência é o tipo de `e2`.

Regra formal:

$$\frac{\Gamma \vdash e_1 : \text{Unit} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T} \quad \text{(T-Sequence)}$$

A restrição de que `e1` deve ser `Unit` garante que apenas comandos com efeitos colaterais (como atribuições, loops, etc.) possam aparecer na primeira posição. Valores como inteiros ou booleanos não são permitidos.

Avaliação

A avaliação segue a ordem estrita da esquerda para direita:

1. Avaliamos `e1` até obter `Unit`.
2. Avaliamos `e2` e retornamos seu valor.

Regras formais:

Avaliar o primeiro comando:

$$\frac{e_1 \to e_1'}{e_1; e_2 \ \to \ e_1'; e_2} \quad \text{(E-Seq Step)}$$

Quando o primeiro comando é `Unit`:

$$\frac{}{\text{Unit}; e_2 \ \to \ e_2} \quad \text{(E-Seq)}$$

Se `e1` avaliar para um valor diferente de `Unit`, ocorre erro de tipo em tempo de execução.

Declaração Local `let x : T = e1 in e2`

Sintaxe: `Let (x, T, e1, e2)`
Semântica: **Vincula** o identificador `x` ao valor de `e1` e avalia `e2` no contexto estendido. O comportamento difere se `T` for tipo referência (`ref T'`) ou tipo não-referência.

Inferência de Tipo

Para um `let` ser bem tipado:

1. A expressão `e1` deve ter tipo `T` no ambiente atual.
2. O corpo `e2` é avaliado no ambiente estendido com `x : T`.
3. O tipo do `let` é o tipo de `e2` no ambiente estendido.

Regra formal:

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

A inferência de tipos em L2 inclui:

- Para **tipos não-referência** (`int` , `bool` , `unit`): O tipo de `e1` deve coincidir exatamente com `T` .
- Para **tipos referência** (`ref T`): O tipo de `e1` também deve ser `ref T` .

Avaliação

A avaliação segue duas estratégias distintas:

Para tipos não-referência:

1. Avaliamos `e1` até obter um valor `v` .
2. Substituímos todas as ocorrências livres de `x` em `e2` por `v` .
3. Avaliamos a expressão resultante.

Regras:

$$\frac{e_1 \to e_1'}{\text{let } x : T = e_1 \text{ in } e_2 \;\to\; \text{let } x : T = e_1' \text{ in } e_2} \quad \text{(E-Let-Step)}$$

$$\frac{\text{valor}(v_1) \quad T \neq \text{Ref } T'}{\text{let } x : T = v_1 \text{ in } e_2 \;\to\; e_2\{v_1/x\}} \quad \text{(E-Let)}$$

Para tipos referência:

1. Avaliamos `e1` até obter um valor `VLocation(l)` .
2. Estendemos a tabela de símbolos para associar `x` à localização `l` .
3. Avaliamos `e2` com esta nova associação.

Regra:

$$\frac{\text{valor}(VLocation(l)) \quad T = \text{Ref } T'}{\text{let } x : T = VLocation(l) \text{ in } e_2 \;\to\; e_2 \quad (\text{com } x \mapsto l \text{ no ambiente})} \quad \text{(E-Let-Ref)}$$

Se `e1` não for uma localização quando `T` é referência, ocorre erro de tipo em tempo de execução.

Criação de Referência (new e)

Sintaxe: `New e`
Aloca uma nova posição l na memória, armazena o valor de `e` nessa posição e retorna a localização l.

Inferência de Tipo

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \text{new } e : \text{Ref } T} \text{(T-New)}$$

Avaliação

A expressão e pode ser qualquer termo válido: um valor, variável, ou expressão complexa. O tipo de e determina o tipo da referência criada.
Avaliação

A avaliação segue a ordem:

1. Avaliamos `e` até obter um valor `v`.
2. Alocamos uma **nova** posição l na memória σ.
3. Armazenamos `v` na célula l.
4. Retornamos a localização l como valor `VLocation l`.

Regras formais:
Enquanto `e` não é um valor, avalie-o até que seja.

$$\frac{e \to e'}{\text{new } e \ \to \ \text{new } e'} \quad \text{(E-New Step)}$$

Quando `e` for um valor, alocar nova posição na memória:

$$\frac{l \notin \text{Dom}(\sigma)}{\text{new } e,\ \sigma \ \to \ \text{VLocation l},\ \sigma[l \mapsto v]} \quad \text{(E-New 1)}$$

Dereferência (desreferenciação) `!e`

Sintaxe: `Dereference e`
Acessa o valor armazenado na localização referenciada `e`.
A expressão e deve ser uma referência.

Inferência de Tipo

Para que `!e` seja bem tipado, é necessário que

1. `e` seja tipado em `ref t`
2. o tipo de `!e` seja `t`.

$$\frac{\Gamma \vdash e : \text{Ref } T}{\Gamma \vdash !e : T} \text{(T-Deref)}$$

Avaliação

A avaliação segue os passos:

1. Avaliamos `e` até obter uma localização `VLocation(l)`.
2. Consultamos a memória no endereço l para obter o valor `v` armazenado.
3. Retornamos `v`.

Regras formais:

$$\frac{e \to e'}{!e \ \to \ !e'} \text{(E-Deref Step)}$$

$$\frac{l \in \text{Dom}(\sigma) \quad \wedge \quad \sigma(l) = v}{!\text{VLocation}(l) \ \to \ v} \text{(E-Deref 1)}$$

Se a localização l não existir na memória, ocorre erro de execução análogo a `EvaluationError("Localização inválida")`. Se e não for uma localização, também ocorre erro.

```
Dataview (inline field '='): Error:
-- PARSING FAILED ------------------------------------------------
```

```
> 1 | =
    | ^

Expected one of the following:

'(', 'null', boolean, date, duration, file link, list ('[1, 2, 3]'), negated
field, number, object ('{ a: 1, b: 2 }'), string, variable
```