# Inferência de Tipos, Equivalência e Módulos

Linguagens de Programação

16 de abril de 2019

# Tópicos

# Inferência de Tipos

## Linguagem Estaticamente Tipada

- **Toda** associação (binding) tem um **tipo** que é determinado em **tempo de compilação**. (Ex. ML, Java, C)
- Em contra-posição à Linguagem Dinamicamente Tipada (tipo é determinado apenas em tempo de execução). (Ex. Racket, Ruby, Phyton)

## Inferência de Tipos, Verificação de Tipos, Polimorfismo

- São conceitos separados
- Qual o tipo da declaração ou da expressão?
- Tipos de partes da declaração (expressão) são compatíveis?
- A declaração (expressão) pode ser usada consistentemente para mais de um tipo?

# Como inferência de tipos funciona em ML?

## Visão Geral

- Determina sequencialmente o tipo das associações. Usa tipos já descobertos para descobrir os seguintes.
- Para cada associação `val`, `fun`, etc são analisados fatos sobre o respectivo tipo (ex. `x + 1`, `int + int`, portanto, `x: int`)
- Se não existe nenhuma restrição de tipo, é usada uma variável de tipo (Ex., 'a)

# Exemplo 1

```
val x = 42;
fun f (y,z,w)=
  if y
  then
       z+x
  else 0
```

x: int, pois 42 é int
f: T1 -> T2
T1 = T3 * T4 * T5
  y: T3, z: T4, w: T5
T3 = bool, pois y na condição
T4 = int, pois z + x e x é int
T2 = int
*renomeando consistentemente* T5 => 'a

**f: bool * int * 'a -> int**

# Exemplo 2

```
fun f x =                (* i   *)
  let
      val (y,z) = x      (* ii  *)
  in
      (abs y) + z        (* iii *)
  end
```

```
f: T1 -> T2
por ii, T1 = T3 * T4, y: T3, z: T4
por iii:
  abs: int -> int
  portanto, y e z são int
  assim T3=int e T4: int
```

**f: int * int -> int**

# Exemplo 3

```
fun sum xs =
 case xs of
    [ ] => 0                    (* i *)
  | x::xs' => x + (sum xs')(* ii *)
```

```
sum: T1 -> T2
por i, T1 = T3 list
por ii:
    - x + sum xs' => T2 = int
    - T3 = int
```

**sum: int list -> int**

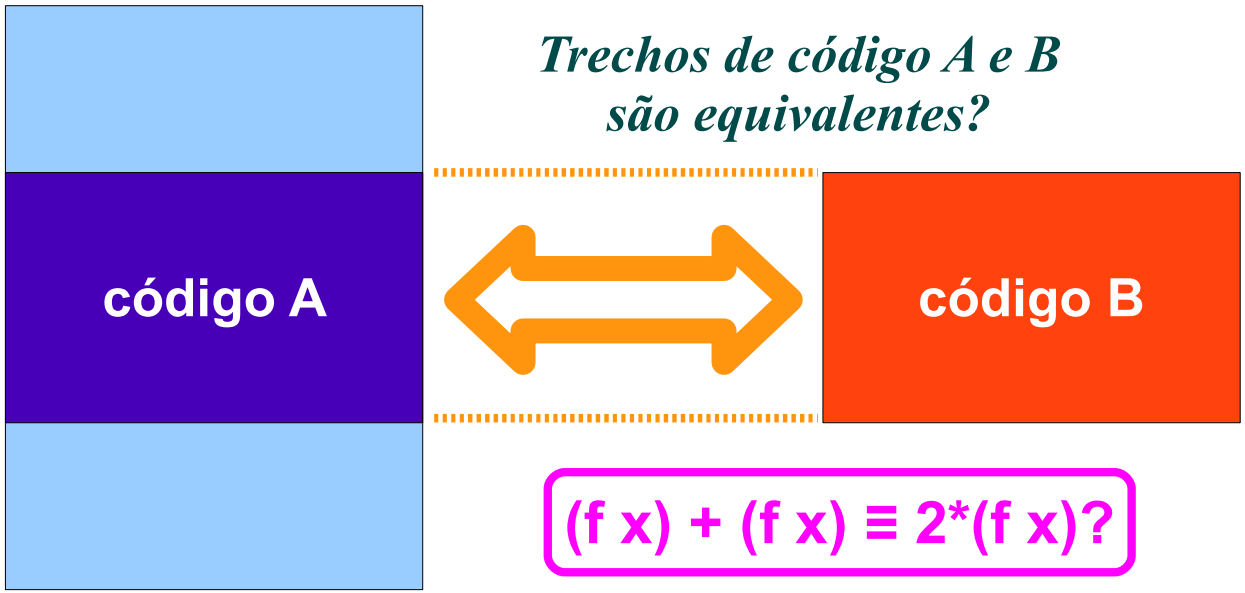
# Exemplo 4

```
fun length xs =
 case xs of
    [ ] => 0                          (* i  *)
  | _::xs' => 1 + (length xs')(* ii *)
```

```
length: T1 -> T2
por i, T1 = T3 list
por ii:
 - 1 + length xs' => T2 = int
renomeando T3 => 'a
```

**length: 'a list -> int**

# Equivalência de Código




## Cenários em que equivalência é considerada:

- Manutenção: simplificar, reorganizar
- Compatibilidade com versões anteriores: acrescentar novas características sem mudar as características existentes
- Otimização: código mais rápido ou com menos uso de memória
- Abstração: mudança observada por cliente externo?

# Equivalência de Funções

## Duas funções, ao receberem os mesmos argumentos, dentro do mesmo ambiente são equivalentes se:

- Produzem o mesmo resultado
- terminam da mesma forma: se um executa indefinidamente, o outro também o fará
- Modifica a mesma memória visível aos clientes da mesma forma
- Produz a mesma E/S
- Levanta as mesmas exceções

## Ou seja, sob os mesmos argumentos:

- Produz os mesmos resultados
- Produz os mesmos efeitos colaterais

# Benefícios de programar sem efeitos colaterais

## Evitar a repetição de cálculos

`(f x) + (f x)` é equivalente a `2*(f x)`

## Reordenar expressões: g(x) - f(x)

```
int a = f(x);
int b = g(x);
return b - a;
```

```
signature RATIONAL =
sig
    datatype rational = Frac of int * int;
    exception BadFrac;
    val make_frac : int * int -> rational;
    val add : rational * rational -> rational;
    val toString : rational -> string;
end
1\---  rational.sig    All L7      (SML)
structure Rational :> RATIONAL =
struct
    datatype rational = Frac of int * int;
    exception BadFrac;
    fun make_frac (n,d) = Frac (n,d);
    fun add (Frac(n1,d1),Frac(n2,d2)) = Frac (d2*n1 + d1 *n2,d1*d2)
    fun toString (Frac(n,d)) = (Int.toString n) ^ "/" ^ (Int.toString d) ;
end
1\**-  rational.sml    All L5      (SML)
structure Rational : RATIONAL
val it = () : unit
- val r1 = Rational.Frac(1,2);
val r1 = Frac (1,2) : Rational.rational
-
```