

Linguagens e Ambientes de Programação (Aula Teórica 6)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Agenda

- Funções recursivas sobre números naturais.
- Raciocínio indutivo vs. Raciocínio iterativo.
- Tail recursion

Inferência de tipos (Recap Exercícios)

- Exercícios, que soluções existem para o tipo destas expressões?

```
fun x y z -> if x then y else z
```

```
fun w -> if w then  
    fun x y -> x = y  
else  
    fun x y -> x <> y
```

Funções recursivas

- A recursão é um mecanismo que permite definir uma entidade (função, tipo, classe, etc.), usando na sua definição o seu próprio nome.

```
let rec x = e1 in e2
```

```
class Node<T> {  
    T value;  
    Node<T> next;  
}
```

- A recursão é um mecanismo que permite instanciar o mesmo código (função) mais que uma vez no mesmo traço de execução, com valores potencialmente diferentes para os parâmetros.

Funções recursivas

- As linguagens funcionais utilizam a recursão como meio básico para iterar.

```
let rec loop () = read_int () |> print_int; print_endline ""; loop ();;
```

- A utilização recursiva de um nome tem que estar sempre “guardada” pela definição de uma função.

Sintaxe Hint: Operadores

▷ `let (|>) x f = f x`

`let rec loop () = read_int () |> print_int; print_endline "";; loop ();;`

`let (^^) x y = max x y`

`(+)`

`- : int -> int -> int = <fun>`

Funções recursivas (exemplo: somatório)

- A forma geral da soma está descrita pelo somatório de um limite inferior (l) a um limite superior (u) onde se aplica uma função (f). Esta abordagem em ordem superior permite abstrair a produção de elementos.

$$\sum_{i=l}^u f(i)$$

```
let rec sum f l u =  
  if l > u then 0  
  else f l + sum f (l+1) u
```

```
utop # sum (fun x -> 2*x) 1 10  
;;  
- : int = 110
```


Execução baseada numa pilha

- A chamada de funções é baseada numa pilha, onde são guardados os valores dos parâmetros para cada chamada e as variáveis locais correspondentes.
- Cada chamada ocupa espaço em memória.

```
let rec sum n =  
  if n = 0 then 0  
  else n + sum (n-1)
```

```
utop # sum 10;;  
sum <-- 10  
sum <-- 9  
sum <-- 8  
sum <-- 7  
sum <-- 6  
sum <-- 5  
sum <-- 4  
sum <-- 3  
sum <-- 2  
sum <-- 1  
sum <-- 0  
sum --> 0  
sum --> 1  
sum --> 3  
sum --> 6  
sum --> 10  
sum --> 15  
sum --> 21  
sum --> 28  
sum --> 36  
sum --> 45  
sum --> 55  
- : int = 55
```


Execução baseada numa pilha

- A chamada de funções é baseada numa pilha, onde são guardados os valores dos parâmetros para cada chamada e as variáveis locais correspondentes.
- Cada chamada ocupa espaço em memória.

```
let rec count n =  
  if n = 0 then 0  
  else 1 + count (n-1)
```

```
utop # count 100000;;  
- : int = 100000
```

```
utop # count 1000000;;  
Stack overflow during evaluation (looping recursion?).
```

Execução baseada numa pilha

- A chamada de funções é baseada numa pilha, onde são guardados os valores dos parâmetros para cada chamada e as variáveis locais correspondentes.
- Cada chamada ocupa espaço em memória.

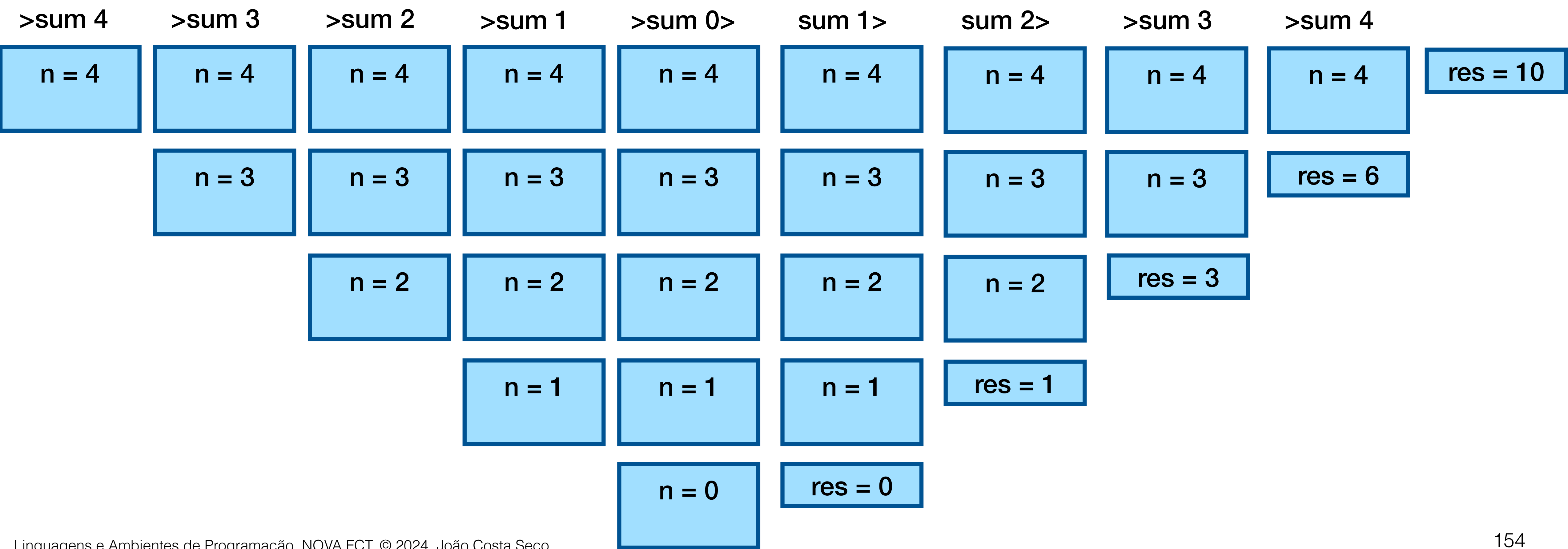
```
class Main {
    static int count(int n) {
        if (n == 0) {
            return 0;
        }
        return 1+count(n-1);
    }

    public static void main(String[] args) {
        System.out.println(count(20000));
    }
}
```

[illegible]

A pilha de execução e as chamadas de função

- Quando uma função tem alguma computação para fazer entre a chamada recursiva e o final, e devolver o resultado, então precisa de manter todos os registos de ativação.



“Tail recursion”

- A mesma função pode ser escrita de outra forma de modo a que, a seguir à chamada recursiva, não haja mais computação nenhuma.

```
let rec sum n =  
  if n = 0  
  then 0  
  else n + sum(n-1)
```

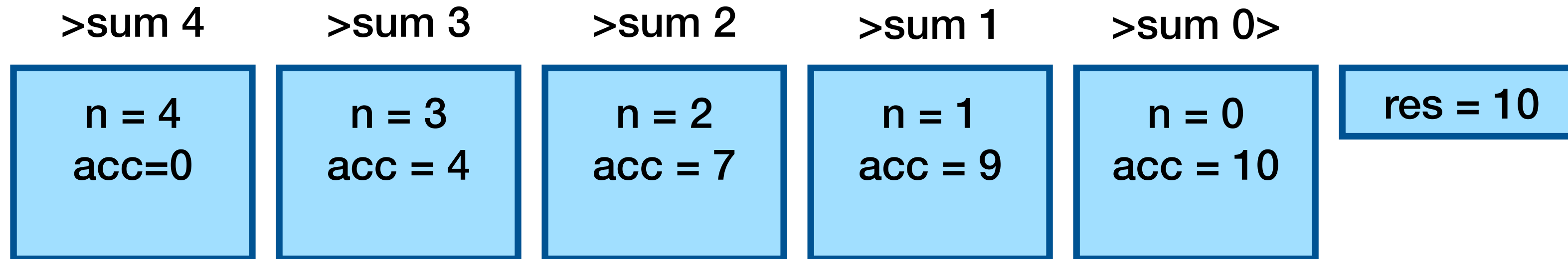
```
let sum n =  
  let rec sum' n acc =  
    if n = 0  
    then acc  
    else sum' (n-1) (n+acc)  
  in sum' n 0
```

```
utop # count 1000000;;  
Stack overflow during evaluation (looping recursion?).
```

```
utop # sum 1000000;;  
- : int = 500000500000
```

“Tail recursion” e a pilha de execução

- Se a chamada recursiva é a ultima coisa a fazer na função, o registo de ativação pode ser reutilizado porque as variáveis locais não vão ser precisas depois de retornar e o resultado já está no sítio (topo da pilha).



Funções indutivas nos números naturais (correção)

- Funções recursivas podem seguir um raciocínio indutivo para chegar ao resultado. É possível provar a sua correção através de uma hipótese de indução.

```
(** [sum n] is the sum of the first [n] positive integers
    requires: [n >= 0] *)
let rec sum n =
  if n = 0
  then 0                (* base case: sum 0 = 0 *)
  else n + sum(n-1)      (* inductive case: sum n = n + sum(n-1) *)
```

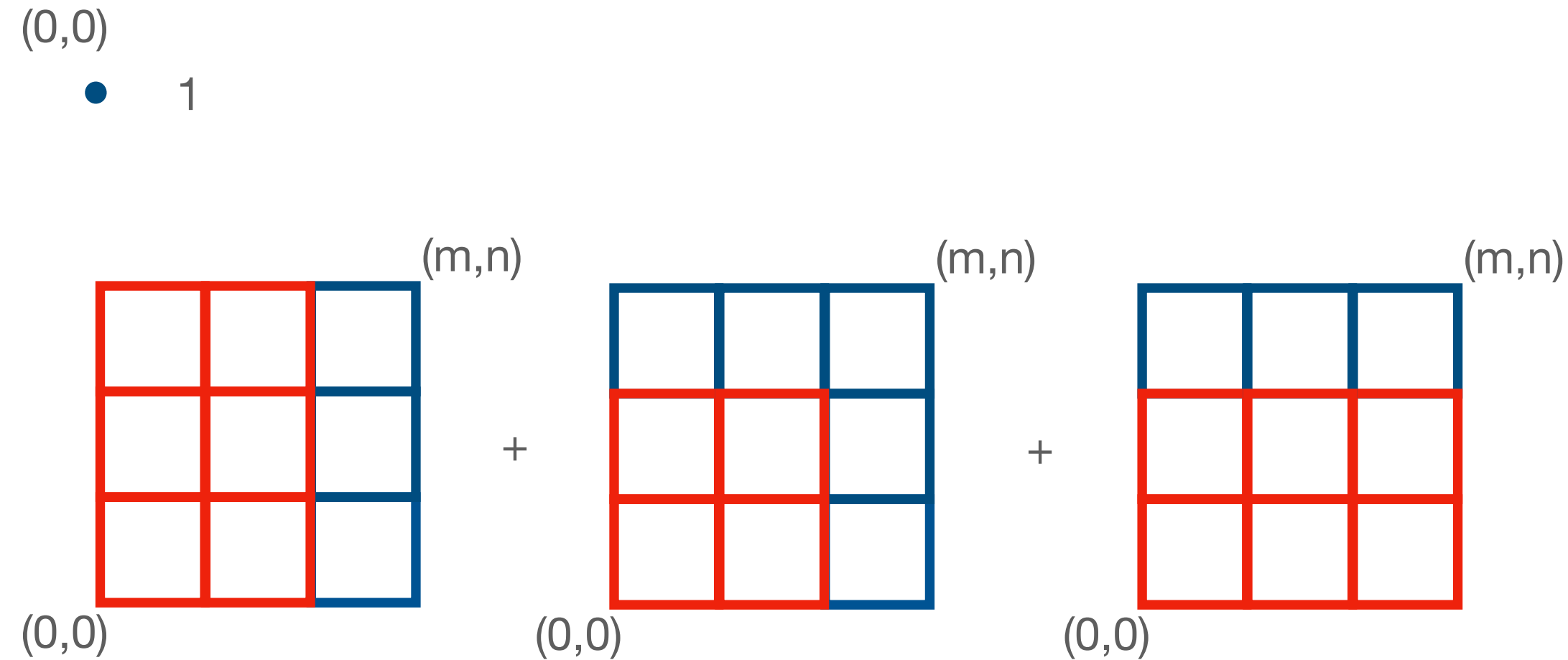
Funções indutivas nos números naturais (tail recursion)

- Funções recursivas podem seguir um raciocínio indutivo para chegar ao resultado. É possível provar a sua correção através de uma hipótese de indução.

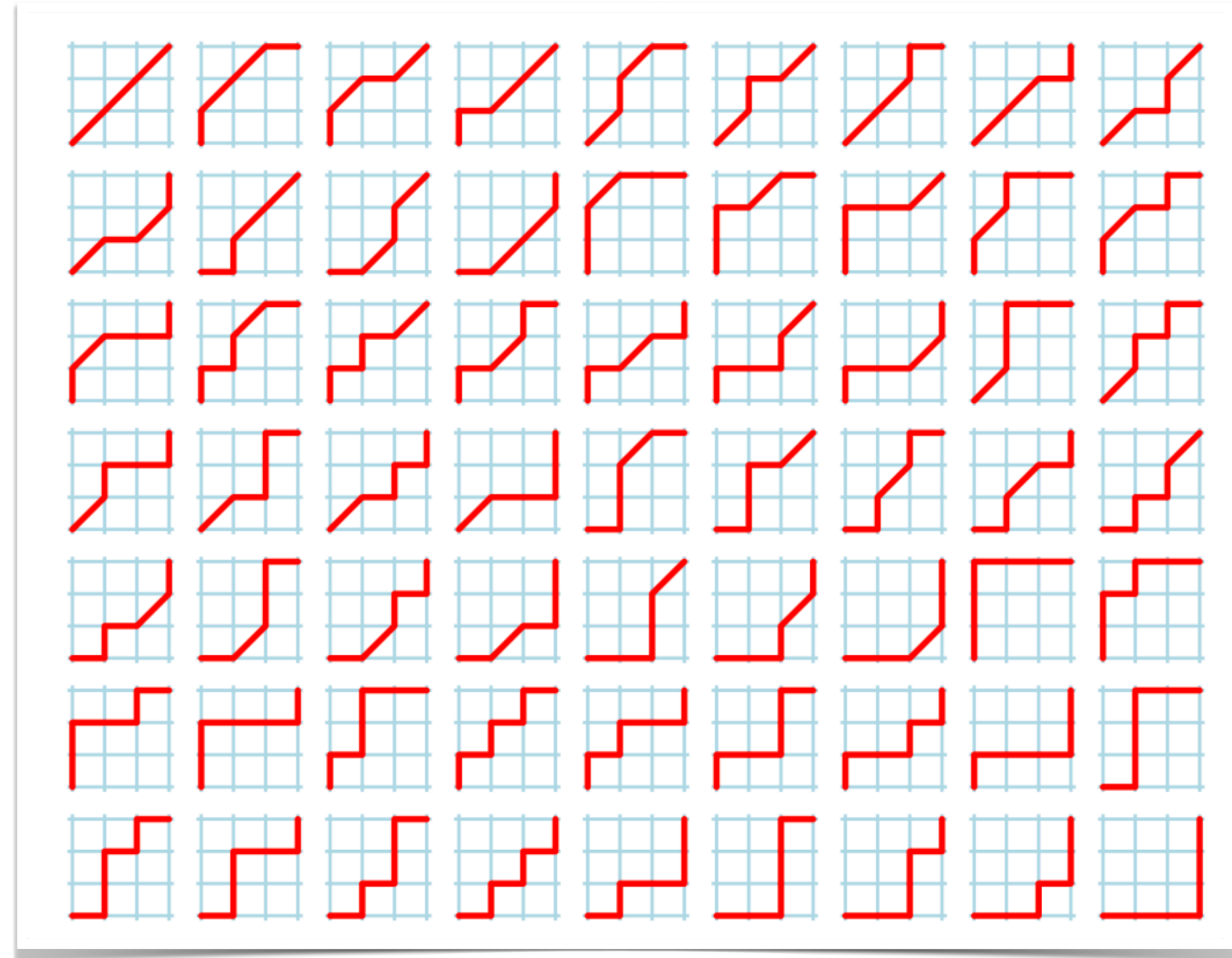
```
let sum m =  
  let rec sum' n acc =  
    if n = 0  
    then acc  
    else sum' (n-1) (n+acc)  
  in sum' m 0  
(* post-condition: sum' n acc = sum m && acc = (sum m) - (sum n) *)  
(* base case: sum' 0 acc = sum m && acc = (sum m) - (sum 0) *)  
(* inductive case: sum' (n-1) (n+acc) = sum m  
  && n+acc = (sum m) - (sum (n-1))  
  ==> acc = (sum m) - (n + sum (n-1))  
  ==> acc = (sum m) - (sum n). qed. *)  
(* conclusion: sum' m 0 = sum m && acc = (sum m) - (sum m) = 0 *)
```


Funções indutivas: o número de Delannoy

- Determine o número de caminhos que existem numa grelha de n por m , entre o ponto $(0,0)$ e o ponto (m,n) usando passos de unidade 1 no sentido “norte”, “nordeste”, e “este”.

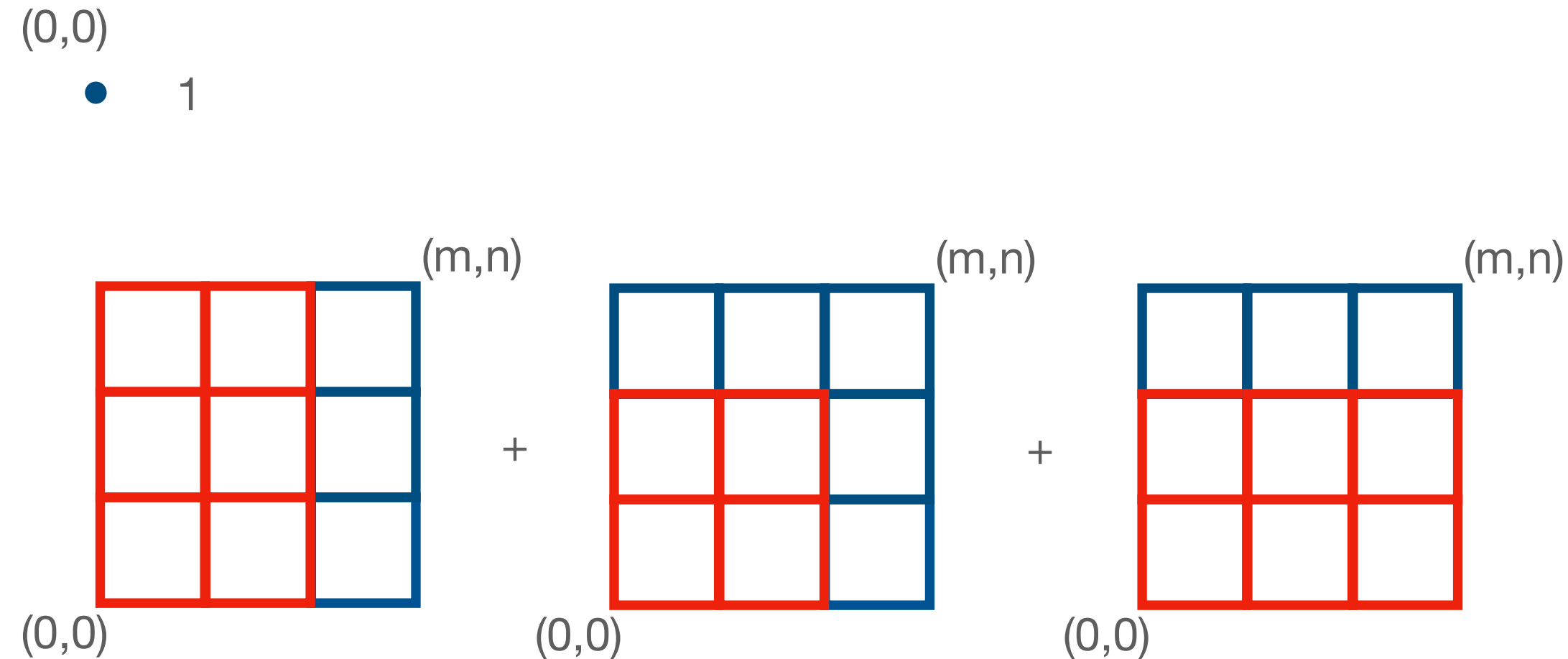


https://en.wikipedia.org/wiki/Delannoy_number



Funções indutivas: o número de Delannoy

- Determine o número de caminhos que existem numa grelha de n por m , entre o ponto $(0,0)$ e o ponto (m,n) usando passos de unidade 1 no sentido “norte”, “nordeste”, e “este”.



```
let rec delannoy m n =  
  if m = 0 || n = 0 then 1  
  else delannoy (m-1) n + delannoy m (n-1) + delannoy (m-1) (n-1)
```

✓ 0.0s

```
val delannoy : int → int → int = <fun>
```

