Tail Recursion

Zheng Guo

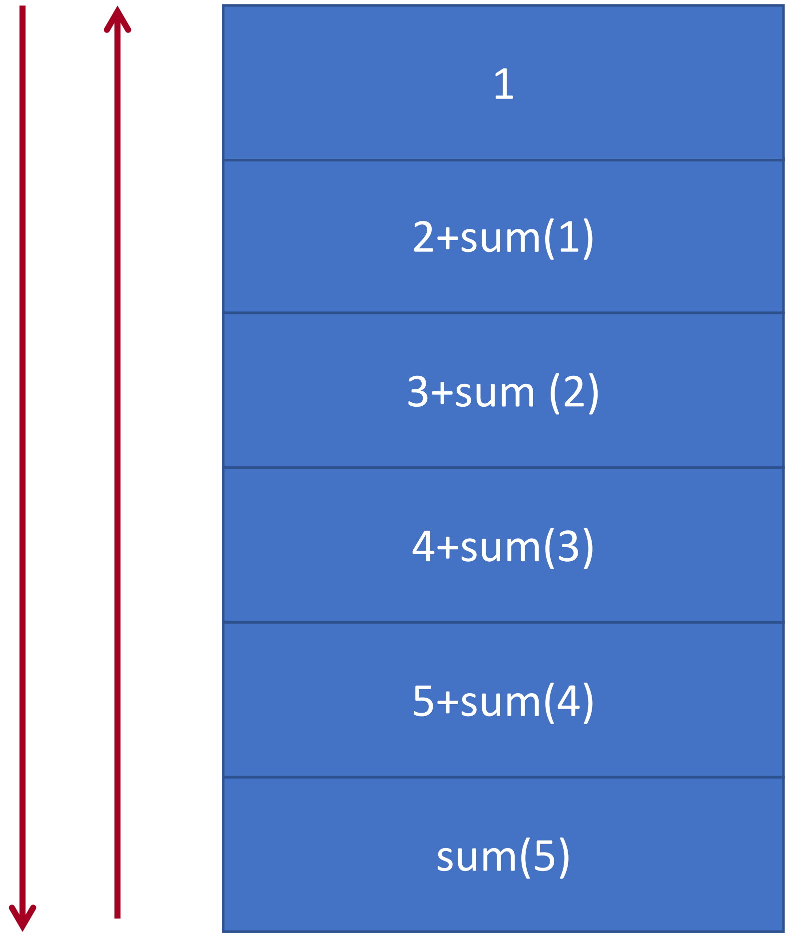
2018/10/10

Agenda

- Tail call
- Examples of tail recursion
- Preview of map

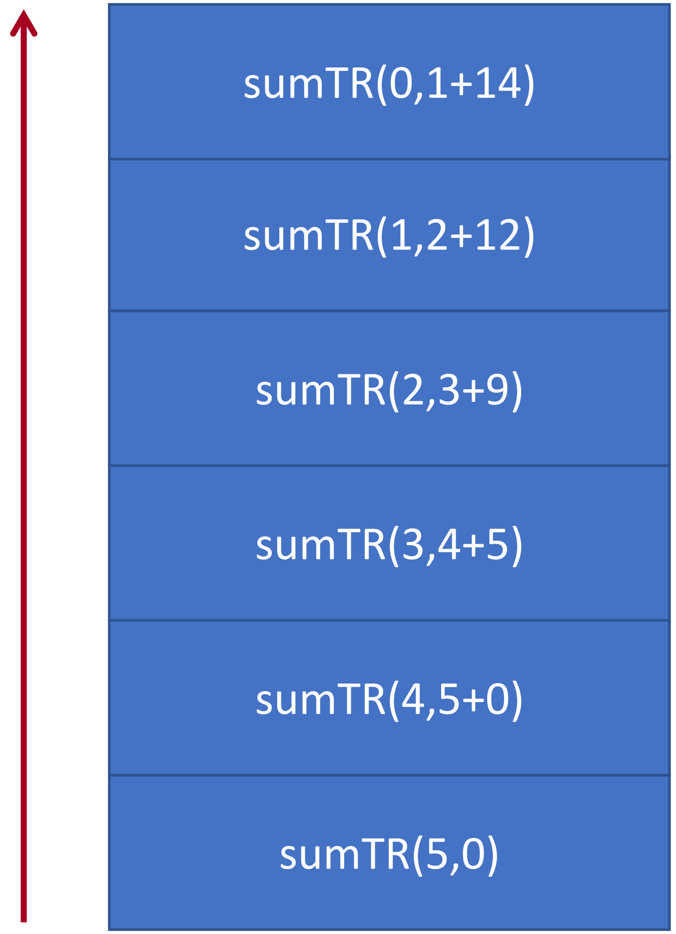
Normal recursion

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



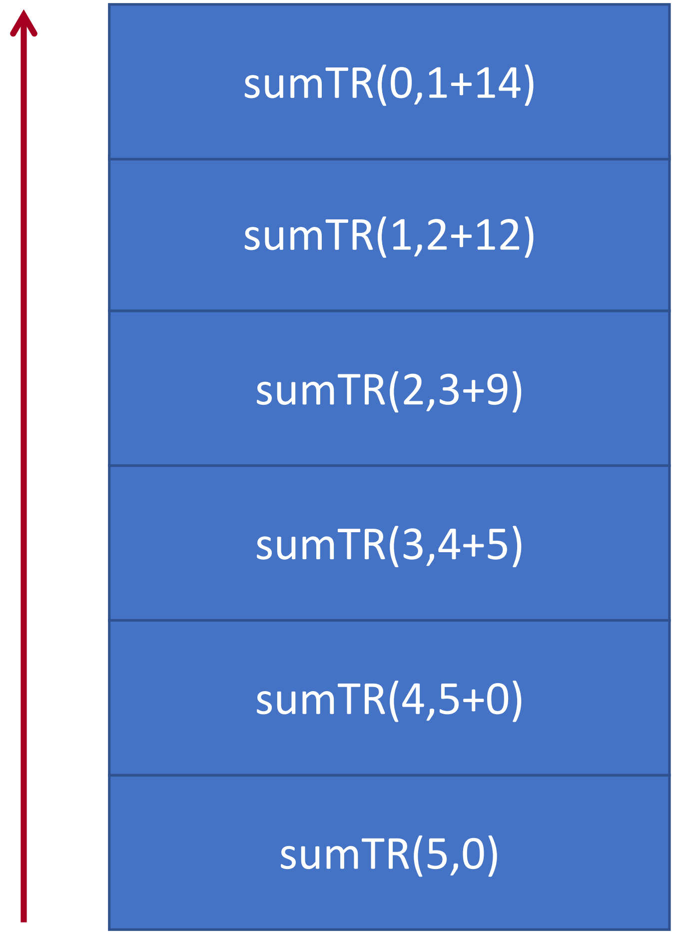
Tail recursion

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



Tail recursion

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



Why tail recursion

- Compiler is SMART!
- Tail recursions are optimized into loops to save memory and time!

Tail call

- Tail call: the resulting value is immediately returned (no further computation is performed on it by the recursive caller)

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

There is addition here!

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

Tail call

- Tail call: the resulting value is immediately returned (no further computation is performed on it by the recursive caller)
  - let rec f p = f p′
  - let rec f p = **if** *cond* **then** f p_1 **else** f p_2
  - let rec f p = **let** b_1 … b_n **in** f p′
  - let rec f p = **match** *e* **with** $case_1$ -> f p_1 | $case_2$ -> f p_2 …

Is this a tail call?

Let f be a recursive function

```
(a) f x y
(b) (f x y)  * 2
(c) f (f x y) z
(d) if y < z then f x y else z
(e) match x with
      | []     -> f 0  []
      | hd::tl -> f hd tl
```

Is this a tail call?

Let f be a recursive function

```
✓ (a) f x y
✗ (b) (f x y)  * 2
✗ (c) f (f x y) z
✓ (d) if y < z then f x y else z
✓ (e) match x with
          | []     -> f 0  []
          | hd::tl -> f hd tl
```

Write a tail recursion

- Create a helper function that takes accumulators
- Old base case becomes initial accumulator
- New base case becomes final accumulator

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

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

Write a tail recursion

- Create a helper function that takes accumulators
- Old base case becomes initial accumulator
- New base case becomes final accumulator

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

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

Example: sum a list of int

sumList : int list -> int

```
let rec sumList xs = match xs with
  | []     -> 0
  | hd::tl -> hd + sumList tl

let rec sumList xs =
  let rec sumListTR xs acc = match xs with
    | []     -> acc
    | hd::tl -> sumListTR tl (hd + acc)
  in sumListTR xs 0
```

Tail call annotation

```
let rec sum n =
  if n <= 1
    then 1
    else n + (sum[@tailcall]) (n-1)
```

This assertion checks whether this function call is a tail call, if not the compiler gives you a warning.

Example: make a list with n copys of the element x

replicate : ‘a -> int -> ‘a list

```
let rec replicate x n =
  if n <= 0 then []
            else x::replicate (n-1) x

let rec replicate x n =
  let rec replicateTR x n acc =
    if n <= 0 then acc
              else replicateTR x (n-1) (x::acc)
  in replicateTR x n []
```

Example: remove odd numbers

```
removeOdds : int list -> int list

let rec removeOdds xs = match xs with
  | []     -> []
  | hd::tl -> if hd mod 2 = 0 then hd::removeOdds tl
                              else removeOdds tl

let removeOdds xs =
  let rec removeOddsTR xs acc =
    match xs with
    | [] -> List.rev acc
    | hd::tl -> if hd mod 2 = 0 then removeOddsTR tl (hd::acc)
                                else removeOddsTR tl (acc)
  in removeOddsTR xs []
```

Example: list partition

partition : int -> int list -> (int list, int list)

```
let rec partition x xs = match xs with
  | []     -> ([], [])
  | hd::tl -> let (l,r) = partition x tl in
              if hd <= x then (hd::l,r) else (l,hd::r)

let partition x xs =
  let rec partitionTR x xs lacc racc = match xs with
    | []     -> (List.rev lacc, List.rev racc)
    | hd::tl -> if hd <= x then partitionTR x tl (hd::lacc) racc
                           else partitionTR x tl lacc (hd::racc)
  in partitionTR x xs [] []
```

Example:

[; ; ;]

name

["John Muir"; "Revelle" ; "Thursgood Marshall";"Earl Warren"]

```
getCollegeNames xs =
  match xs with
    | []     -> []
    | hd::tl -> (name hd)::(getCollegeNames tl)
```

Example:

[ ;  ;  ; ]

firstname

[“Rick” ; “Sorin” ; “Pradeep” ; “Leo”]

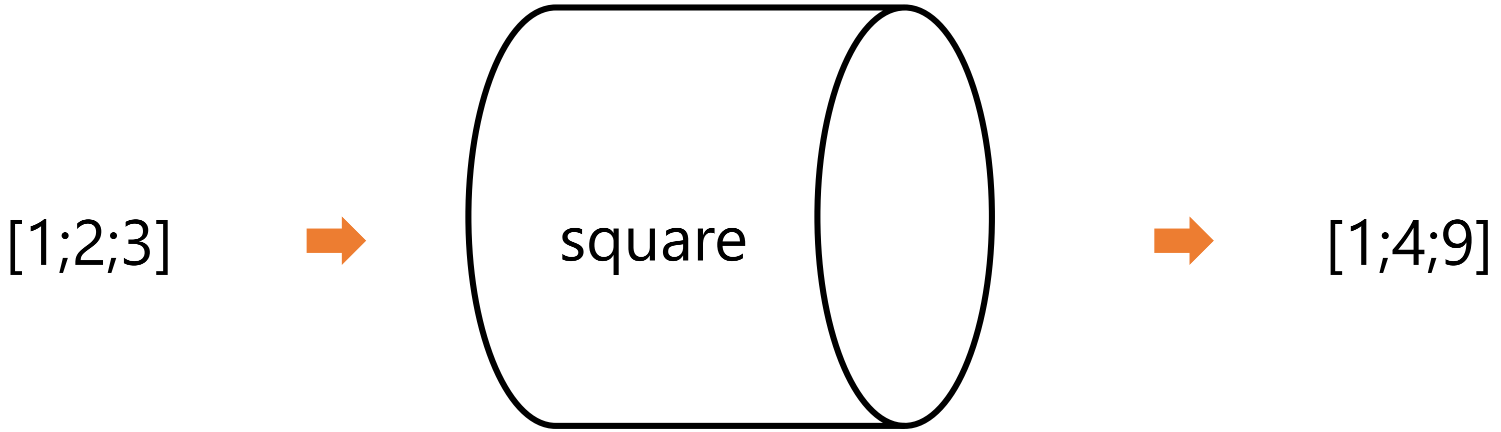
```
getFirstNames xs =
  match xs with
    | []     -> []
    | hd::tl -> (firstname hd)::(getFirstNames tl)
```

```
getCollegeNames xs =
  match xs with
    | []     -> []
    | hd::tl -> (name hd)::(getCollegeNames tl)
```

```
getFirstNames xs =
  match xs with
    | []     -> []
    | hd::tl -> (firstname hd)::(getFirstNames tl)
```

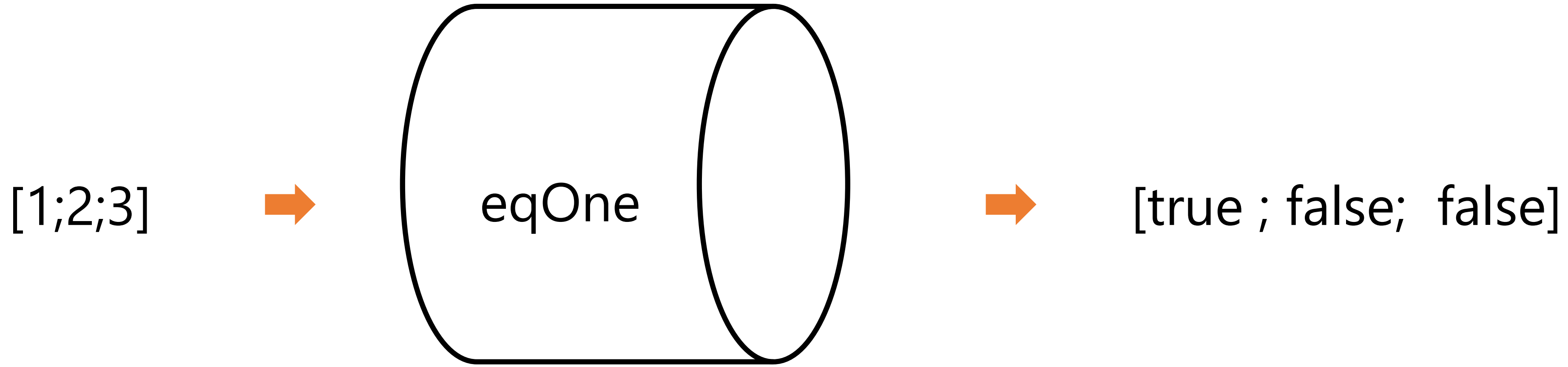
```
map f xs =
  match xs with
    | []     -> []
    | hd::tl -> (f hd)::(map f tl)
```

Map



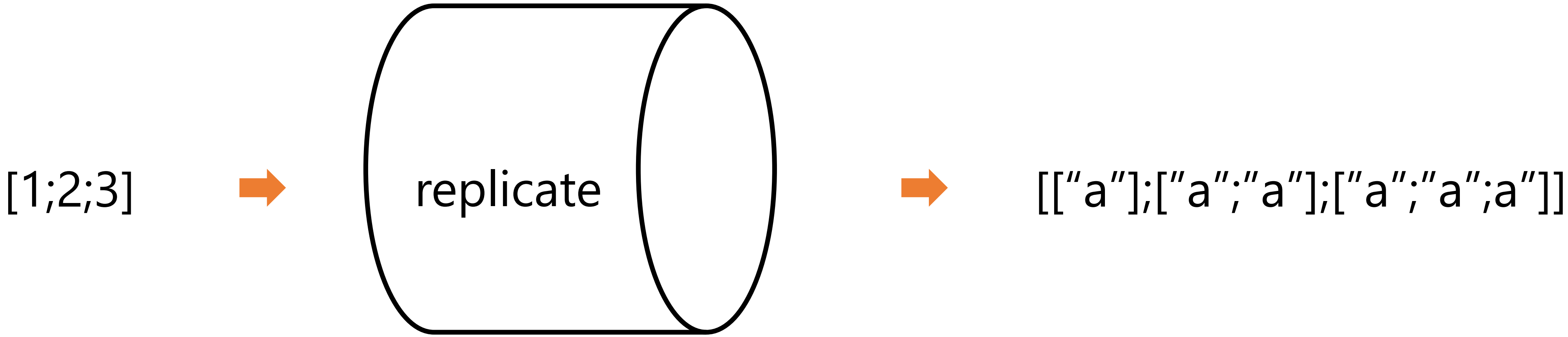

```
let square x = x * x;;
map square [1;2;3];;
```

Map




```
let eqOne = (=) 1;;
map eqOne [1;2;3];;
```

Map




```
let f = replicate “a”;;
map f [1;2;3];;
```

- More about map next time!