# Lecture 11: Parametric polymorphism

↳ same code on different types

+

Datatypes

---

Ad-hoc polymorphism: same name for different code for different types

an int list is
[]
x :: xs, x : int ?
xs : int list
→ and that's it!

a string list is
[]
x :: xs, x : string
xs : string list
→ and that's it!

an 'a list is
[]
x :: xs where x : 'a
xs : 'a list
→ and that's it!

type variable

'a = α
'b = β

# Parametric polymorphism

Parametric → same code for

polymorphism → many shapes

---

```
fun length(l: int list) : int =
  case l of
    [] => 0
  | x::xs => 1 + length xs

length [1,2,3] = 3
```

```
fun length(l: string list) : int =
  case l of
    [] => 0
  | x::xs => 1 + length xs

length ["a", "b"] = 2
```

```
fun length(l: 'a list) : int =
    case l of
        [] => 0
      | _ :: xs => 1 + length xs
```

length [1, 2, 3] = 3 ] 'a = int

length ["a", "b"] = 2 ] 'a = string

```
fun sum(l: intlist) : int =        ('a)
  case l of
    [] => 0
  | x :: xs => x + sum xs
   :'a  :'a list         'a list
```

type error

$x + y$ : int
(x : int, y : int)

fun zip(l₁ : ~~int~~ 'a list, l₂ : ~~string~~ 'b list) : (~~int~~ 'a * ~~string~~ 'b) list =

```
fun zip(l1 : 'a list, l2 : 'b list) : ('a * 'b) list =
    case (l1, l2) of
        ([], _) => []
      | (_, []) => []
      | (x::xs, y::ys) => (x, y) :: zip(xs, ys)
```

In the last pattern: x : 'a, xs : 'a list, ys : 'b list; in the result: x : 'a, y : 'b, xs : 'a list, ys : 'b list

zip([1, 2, 3], ["a", "b", "c"])

[1, 2, 3] : int list, with int = 'a

["a", "b", "c"] : string list, with string = 'b

~~type error~~

fun append(l: 'a list, r: ~~'b~~ 'a list): ~~'b~~ 'a list =

case l of

[] => r  ~~'b~~ 'a list

| x::xs => x :: append(xs, r)

x: 'a   xs: 'a list   x: 'a   append(xs, r): ~~'b~~ 'a list

'a
'a list } → 'a list

# Type inference

```
fun append(l, r) : 'e = 'a list
    case l of
      [] => r
    | x::xs => x :: append(xs, r)
```

'c = 'a list

'd = 'a list

'c = 'a list

'e = 'f list

'f = 'a

an 'a list is
- []
- x :: xs where x : 'a
xs : 'a list
→ and that's it!

a tree is
- Empty
- Node(l, x, r)
l : tree
x : int
r : tree
→ and that's it!

a boolean is
- true
- false
→ and that's it!

# Datatypes

A traffic light color is
- Red, or
- Yellow, or
- Green

→ and that's it!

datatype color =
Red R
| Yellow | Y
| Green | G

constructor

Values: Red, Yellow, Green

ops: case ______

```
fun next(c: Color): color =
  case c of
      Red => Green
    | Yellow => Red
    | Green => Yellow
```

```
case c of
   R => G
 | Y => R
 | G => Y
```

```
datatype bool =
    true
  | false
```

```
case b of
    true => ...
  | false => ...
```

```
data intlist =
    []
  | :: of int * intlist
```

magic infix

```
x::xs

::(x, xs)
```

```
case l : intlist of
    [] => ______
  | ::(x, xs) => ______
```

```
datatype tree =
    Empty
  | Node of tree * int * tree
```

---

## Parametrized datatypes

```
data intlist =
    []
  | :: of int * intlist
```

```
data stringlist =
    []'
  | ::' of string * stringlist
```

datatype 'a list =
[]
| :: of 'a * 'a list

space

- int list
- string list
- (int * string) list

```
data 'a tree =
    Empty
  | Node of 'a tree * 'a * 'a tree
```