Functional Programming / Lean

Higher-order functions

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2026/27

Higher-order functions

- Functions are first-class citizens in Lean
- A function can be
  - stored in data
  - argument of a (**higher-order**) functions
  - returned from a function

Examples of higher-order functions

Most higher-order functions are polymorphic

```
1 List.map    : (α → β) → List α → List β
2 List.filter : (α → Bool) → List α → List α
```

Examples of higher-order functions

Most higher-order functions are polymorphic

```
List.map    : (α → β) → List α → List β
List.filter : (α → Bool) → List α → List α
```

Example uses

```
#eval List.map even [1, 2, 3, 4]
#eval List.filter even [1, 2, 3, 4]
```

Examples of higher-order functions

Most higher-order functions are polymorphic

```
List.map    : (α → β) → List α → List β
List.filter : (α → Bool) → List α → List α
```

Example uses

```
#eval List.map even [1, 2, 3, 4]
#eval List.filter even [1, 2, 3, 4]
```

Lean: type quantifiers are implicit

```
List.map    {α β} : (α → β) → List α → List β
List.filter {α}   : (α → Bool) → List α → List α
```

Instantiation

Instances of a polymorphic type

Consider

```
List.filter {α}     : (α → Bool) → List α → List α
List.filter  : ∀ {α} , (α → Bool) → List α → List α
```

We may replace (instantiate) α by any type:

```
List.filter (α := Int)  : (Int → Bool) → List Int → List Int
@List.filter Char  : (Char → Bool) → List Char → List Char
```

Instantiation

Instances of a polymorphic type

Consider

```
1 List.filter {α}      : (α → Bool) → List α → List α
2 List.filter  : ∀ {α} , (α → Bool) → List α → List α
```

We may replace (instantiate) α by any type:

```
1 List.filter (α := Int)  : (Int → Bool) → List Int → List Int
2 @List.filter Char  : (Char → Bool) → List Char → List Char
```

Instantiation rule

$$\frac{e : \forall \alpha, t \qquad t' : \text{Type}}{e : t[\alpha \mapsto t']}\ \textsc{Inst}$$

Function types

What's the difference between these types?

```
Int -> Int -> Int
Int -> (Int -> Int)
(Int -> Int) -> Int
```

Function types

What's the difference between these types?

```
Int -> Int -> Int
Int -> (Int -> Int)
(Int -> Int) -> Int
```

How many arguments?

```
1 def pick ( b: Bool): Prod Nat Nat → Nat :=
2   match b with
3   | true => Prod.fst
4   | false => Prod.snd
```

Curried functions

Compare these types

```
1 abbrev T1 := Int -> Int -> Int
2 abbrev T2 := Int × Int -> Int
```

- Both function types take two integers and return one
- `T1` takes the arguments one at a time
- `T2` takes both arguments as a pair

Curried functions

Compare these types

```
1 abbrev T1 := Int -> Int -> Int
2 abbrev T2 := Int × Int -> Int
```

- Both function types take two integers and return one
- `T1` takes the arguments one at a time
- `T2` takes both arguments as a pair

Lean prefers types like `T1`

- A **curried** type, after logician Haskell B. Curry
- Predefined functions `Function.curry` and `Function.uncurry` map between `T1` and `T2` (an isomorphism)

```
1 Function.curry   : (α × β → γ) → α → β → γ
2 Function.uncurry : (α → β → γ) → α × β → γ
```

Typing Function Applications

Typing Rule

$$\textsc{App}\quad \frac{e_1 : t_2 \to t_1 \qquad e_2 : t_2}{(e_1\ e_2) : t_1}$$

Typing Function Applications

Typing Rule

$$\textsc{App}\quad \frac{e_1 : t_2 \to t_1 \qquad e_2 : t_2}{(e_1\ e_2) : t_1}$$

Example

Given

$$\begin{array}{lcl} \mathsf{map} & : & \forall\{\alpha\beta\}, (\alpha \to \beta) \to \mathtt{List}\ \alpha \to \mathit{List}\ \beta \\ \mathsf{even} & : & \mathtt{Int} \to \mathtt{Bool} \end{array}$$

$$\dfrac{\dfrac{\mathsf{map} : \forall\{\alpha\beta\}, (\alpha \to \beta) \to \mathtt{List}\ \alpha \to \mathtt{List}\ \beta}{\mathsf{map} : (\mathtt{Int} \to \mathtt{Bool}) \to \mathtt{List\ Int} \to \mathtt{List\ Bool}} \qquad \mathsf{even} : \mathtt{Int} \to \mathtt{Bool}}{\dfrac{\mathsf{map\ even} :?? \qquad \mathsf{[1,2,3,4,5]} : \mathsf{List\ Int}}{\mathsf{map\ even\ [1,2,3,4,5]} :??}}$$

Typing Function Applications

Typing Rule

$$\textsc{App}\quad \frac{e_1 : t_2 \to t_1 \qquad e_2 : t_2}{(e_1\ e_2) : t_1}$$

Example

Given

$$\begin{array}{lcl} \mathsf{map} & : & \forall\{\alpha\beta\}, (\alpha \to \beta) \to \mathtt{List}\ \alpha \to \mathit{List}\ \beta \\ \mathsf{even} & : & \mathtt{Int} \to \mathtt{Bool} \end{array}$$

$$\dfrac{\dfrac{\mathsf{map} : \forall\{\alpha\beta\}, (\alpha \to \beta) \to \mathtt{List}\ \alpha \to \mathtt{List}\ \beta}{\mathsf{map} : (\mathtt{Int} \to \mathtt{Bool}) \to \mathtt{List\ Int} \to \mathtt{List\ Bool}} \qquad \mathsf{even} : \mathtt{Int} \to \mathtt{Bool}}{\dfrac{\mathsf{map\ even} : \mathtt{List\ Int} \to \mathtt{List\ Bool} \qquad [1,2,3,4,5] : \mathtt{List\ Int}}{\mathsf{map\ even}\ [1,2,3,4,5] : ??}}$$

Typing Function Applications

Typing Rule

$$\frac{e_1 : t_2 \to t_1 \qquad e_2 : t_2}{(e_1\ e_2) : t_1}\ \textsc{App}$$

Example

Given

$$\begin{array}{lcl} \mathsf{map} & : & \forall\{\alpha\beta\}, (\alpha \to \beta) \to \mathtt{List}\ \alpha \to \mathit{List}\ \beta \\ \mathsf{even} & : & \mathtt{Int} \to \mathtt{Bool} \end{array}$$

$$\dfrac{\dfrac{\mathsf{map} : \forall\{\alpha\beta\}, (\alpha \to \beta) \to \mathtt{List}\ \alpha \to \mathtt{List}\ \beta}{\mathsf{map} : (\mathtt{Int} \to \mathtt{Bool}) \to \mathtt{List\ Int} \to \mathtt{List\ Bool}} \qquad \mathsf{even} : \mathtt{Int} \to \mathtt{Bool}}{\dfrac{\mathsf{map\ even} : \mathtt{List\ Int} \to \mathtt{List\ Bool} \qquad [1,2,3,4,5] : \mathtt{List\ Int}}{\mathsf{map\ even}\ [1,2,3,4,5] : \mathtt{List\ Bool}}}$$

Designing a higher-order function

Designing a higher-order function

Two functions on lists

```
1 def sum : (xs : List Nat) → Nat
2   | [] => 0
3   | x :: xs => x + sum xs
```

```
1 def product : (xs : List Nat) → Nat
2   | [] => 1
3   | x :: xs => x * sum xs
```

Designing a higher-order function

Two functions on lists

```
def sum : (xs : List Nat) → Nat
  | [] => 0
  | x :: xs => x + sum xs
```

```
def product : (xs : List Nat) → Nat
  | [] => 1
  | x :: xs => x * sum xs
```

The common pattern

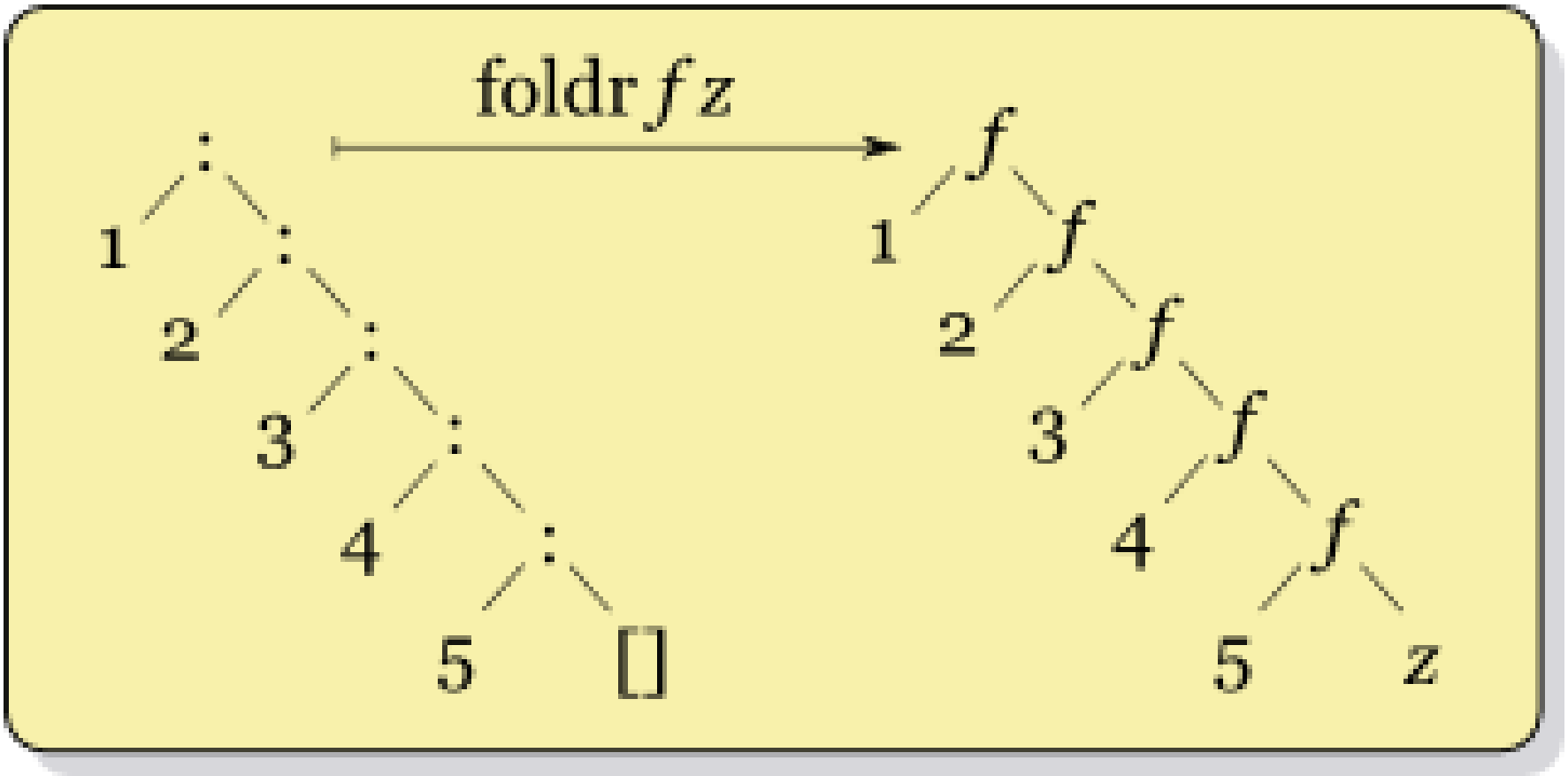
```
def g : (xs : List α) → β
  | [] => z
  | x :: xs => s x (g xs)
```

with parameters

- `z` : α is a value
- `s` : α `->` β `->` β is a combining function

$\Rightarrow$ Represent `g` by some `foldRight f z`

The foldRight function: Intuition



The foldRight function

Making the pattern into a higher-order function

Abstracting over value and combining function

```
1 def foldRight (s : α → β → β) (z : β) : List α → β
2   | [] => z
3   | x :: xs => s x (foldRight s z xs)
```

where

- $\texttt{z} : \beta$ is a value
- $\texttt{s} : \alpha \to \beta \to \beta$ is a combining function

The foldRight function

Making the pattern into a higher-order function

Abstracting over value and combining function

```
1 def foldRight (s : α → β → β) (z : β) : List α → β
2   | [] => z
3   | x :: xs => s x (foldRight s z xs)
```

where

- `z` : β is a value
- `s` : $\alpha \to \beta \to \beta$ is a combining function

What's the type of foldRight?

The foldRight function

Making the pattern into a higher-order function

Abstracting over value and combining function

```
def foldRight (s : α → β → β) (z : β) : List α → β
  | [] => z
  | x :: xs => s x (foldRight s z xs)
```

where

- `z` : β is a value
- `s` : $\alpha \to \beta \to \beta$ is a combining function

What's the type of foldRight?

```
foldRight : ∀ {α β}, (α → β → β) → β → List α → β
```

Foldr in action

sum and product

```
1 def sum' : (xs : List Nat) → Nat := foldRight (· + ·) 0
2 def prod' : (xs : List Nat) → Nat := foldRight (· * ·) 1
```

Foldr in action

sum and product

```
def sum' : (xs : List Nat) → Nat := foldRight (· + ·) 0
def prod' : (xs : List Nat) → Nat := foldRight (· * ·) 1
```

more functions

```
def or_list : List Bool → Bool := sorry
def and_list : List Bool → Bool := sorry
def concat : List (List α) → List α := sorry
def maximum : List Nat → Nat := sorry
```

Transforming functions

Useful operations on functions

- partial application
- operator sections
- function composition
- anonymous functions aka lambda expressions
- eta conversion

Partial application

```
1 #check List.take
2 #check List.take 5
3
4 #check foldRight
5 #check foldRight (· + ·)
6 #check foldRight (· * ·) 1
```

- Partial application = function application with “too few” arguments
- Result is a function
- Can be used like any other function

Operator sections

```
1  -- subtraction
2  (· - ·) : Int -> Int -> Int
3  -- subtract one
4  (· - 1) : Int -> Int
5  -- subtract from one
6  (1 - ·) : Int -> Int
7
8  -- less than 0
9  (· < 0) : Int -> Bool
10 -- greater than 0
11 (0 < ·) : Int -> Bool
```

can be done with every infix function

Function composition

Example

- Remove spaces from string as in this example

```
1 removeWhitespace "abc def \n ghi" == "abcdefghi"
```

Function composition

Example

- Remove spaces from string as in this example

```
1 removeWhitespace "abc def \n ghi" == "abcdefghi"
```

- In module `Char`

```
1 Char.isWhitespace : Char -> Bool
```

Function composition

Example

- Remove spaces from string as in this example

```
1 removeWhitespace "abc def \n ghi" == "abcdefghi"
```

- In module `Char`

```
1 Char.isWhitespace : Char -> Bool
```

- yields definition

```
1 def removeWhitespace (s : String) : String :=
2   (s.toList.filter (not ∘ Char.isWhitespace)).toString
```

Function composition

Example

- Remove spaces from string as in this example

```
removeWhitespace "abc def \n ghi" == "abcdefghi"
```

- In module `Char`

```
Char.isWhitespace : Char -> Bool
```

- yields definition

```
def removeWhitespace (s : String) : String :=
  (s.toList.filter (not ∘ Char.isWhitespace)).toString
```

- Operator "∘" is function composition defined as notation on top of `Function.comp`.

```
def compose {α β γ : Type} (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)
infixr :90 " ∘ " => compose
```

Anonymous functions

- Usual function definition

```
1 def snoc (x : α) (xs : List α) : List α := xs ++ [x]
```

Anonymous functions

- Usual function definition

```
1 def snoc (x : α) (xs : List α) : List α := xs ++ [x]
```

- Alternative: define using an anonymous function aka lambda expression

```
1 def snoc' : α → List α → List α :=
2   fun x xs => xs ++ [x]
```

Anonymous functions

- Usual function definition

```
1 def snoc (x : α) (xs : List α) : List α := xs ++ [x]
```

- Alternative: define using an anonymous function aka lambda expression

```
1 def snoc' : α → List α → List α :=
2   fun x xs => xs ++ [x]
```

- Often for function used in one place as in

```
1 def f3 : List α → List α := foldRight snoc []
```

Anonymous functions

- Usual function definition

```
1 def snoc (x : α) (xs : List α) : List α := xs ++ [x]
```

- Alternative: define using an anonymous function aka lambda expression

```
1 def snoc' : α → List α → List α :=
2   fun x xs => xs ++ [x]
```

- Often for function used in one place as in

```
1 def f3 : List α → List α := foldRight snoc []
```

- Equivalently replace `snoc` by its definition

```
1 def f3' : List α → List α := foldRight (fun x xs => xs ++ [x]) []
```

Anonymous functions

- Usual function definition

```
1 def snoc (x : α) (xs : List α) : List α := xs ++ [x]
```

- Alternative: define using an anonymous function aka lambda expression

```
1 def snoc' : α → List α → List α :=
2   fun x xs => xs ++ [x]
```

- Often for function used in one place as in

```
1 def f3 : List α → List α := foldRight snoc []
```

- Equivalently replace `snoc` by its definition

```
1 def f3' : List α → List α := foldRight (fun x xs => xs ++ [x]) []
```

- Shorten more using partial application

```
1 def f3'' : List α → List α := foldRight (fun x => (· ++ [x])) []
```

Eta conversion

1. A number of definitions have the form

```
1 def f (x : A) : B := g x
```

   where `x` does not occur in `g`

2. In such cases, the formal parameter `x` is redundant:

```
1 def f : A -> B := g
```

   is an equivalent definition **if** `g` is terminating.

3. The transformation from (1) to (2) is called **eta reduction**.[1]
4. The typing of an eta-reduced definition is more restricted.

[1] Reverse transformation: **eta expansion**; both directions: **eta conversion**

Examples for eta-reduced definitions

```
1 def sum' : (xs : List Nat) → Nat := foldRight (· + ·) 0
2 def prod' : (xs : List Nat) → Nat := foldRight (· * ·) 1
```

Wrapup

Higher-order functions

- take functions as parameters,
- often have polymorphic types,
- abstract common patterns (map, filter, foldr),
- enable powerful programming techniques (partial application, operator sections, function composition, anonymous functions, eta conversion).