Functional Programming for Logicians

Péter Mekis

Department of Logic, ELTE Budapest

Session 8: 2019 April 1

Lambda operator in the lambda calculus

- If $f \in \mathsf{Exp}_\beta$ and $x \in \mathsf{Var}_\alpha$, then $(\lambda x \,.\, f) \in \mathsf{Exp}_{\langle \alpha, \beta \rangle}$

Lambda operator in the lambda calculus

- If $f \in \mathsf{Exp}_\beta$ and $x \in \mathsf{Var}_\alpha$, then $(\lambda x \, . \, f) \in \mathsf{Exp}_{\langle \alpha, \beta \rangle}$
- Types: $\mathtt{e} \mid \mathtt{t} \mid \langle \alpha, \beta \rangle$

Lambda operator in the lambda calculus

- If $f \in \mathsf{Exp}_\beta$ and $x \in \mathsf{Var}_\alpha$, then $(\lambda x\,.\,f) \in \mathsf{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\texttt{e} \mid \texttt{t} \mid \langle\alpha, \beta\rangle$
  $\texttt{p1} =_{\mathsf{def}} \langle\texttt{e}, \texttt{t}\rangle$

Lambda operator in the lambda calculus

- If $f \in \mathsf{Exp}_\beta$ and $x \in \mathsf{Var}_\alpha$, then $(\lambda x \,.\, f) \in \mathsf{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\mathtt{e} \mid \mathtt{t} \mid \langle\alpha,\beta\rangle$
  $\mathtt{p1} =_{\mathsf{def}} \langle\mathtt{e},\mathtt{t}\rangle$
- $\mathsf{and}_{\mathsf{pred1}} \in \mathsf{Exp}_{\langle\mathtt{p1},\langle\mathtt{p1},\mathtt{p1}\rangle\rangle}$

Lambda operator in the lambda calculus

- If $f \in \mathrm{Exp}_\beta$ and $x \in \mathrm{Var}_\alpha$, then $(\lambda x\,.\,f) \in \mathrm{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\mathsf{e} \mid \mathsf{t} \mid \langle\alpha,\beta\rangle$
  $\mathsf{p1} =_{\mathsf{def}} \langle\mathsf{e},\mathsf{t}\rangle$
- $\mathsf{and}_{\mathsf{pred1}} \in \mathrm{Exp}_{\langle\mathsf{p1},\langle\mathsf{p1},\mathsf{p1}\rangle\rangle}$
  $\mathsf{and}_{\mathsf{pred1}} =_{\mathsf{def}} (\lambda Y_{\mathsf{p1}}\,.\,(\lambda Z_{\mathsf{p1}}\,.\,(\lambda x_{\mathsf{e}}\,.\,(Yx \wedge Zx))))$

Lambda operator in the lambda calculus

- If $f \in \mathsf{Exp}_\beta$ and $x \in \mathsf{Var}_\alpha$, then $(\lambda x \,.\, f) \in \mathsf{Exp}_{\langle \alpha, \beta \rangle}$
- Types: $\mathsf{e} \mid \mathsf{t} \mid \langle \alpha, \beta \rangle$
  $\mathsf{p1} =_{\mathsf{def}} \langle \mathsf{e}, \mathsf{t} \rangle$
- $\mathsf{and}_{\mathsf{pred1}} \in \mathsf{Exp}_{\langle \mathsf{p1}, \langle \mathsf{p1}, \mathsf{p1} \rangle \rangle}$
  $\mathsf{and}_{\mathsf{pred1}} =_{\mathsf{def}} (\lambda Y_{\mathsf{p1}} \,.\, (\lambda Z_{\mathsf{p1}} \,.\, (\lambda x_{\mathsf{e}} \,.\, (Yx \wedge Zx))))$
- $\mathsf{every}_{\mathsf{pred1}} \in \mathsf{Exp}_{\langle \mathsf{p1}, \langle \mathsf{p1}, \mathsf{t} \rangle \rangle}$

Lambda operator in the lambda calculus

- If $f \in \mathrm{Exp}_\beta$ and $x \in \mathrm{Var}_\alpha$, then $(\lambda x \,.\, f) \in \mathrm{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\mathtt{e} \mid \mathtt{t} \mid \langle\alpha,\beta\rangle$
  $\mathtt{p1} =_{\mathrm{def}} \langle \mathtt{e}, \mathtt{t} \rangle$
- $\mathrm{and}_{\mathrm{pred1}} \in \mathrm{Exp}_{\langle \mathtt{p1}, \langle \mathtt{p1}, \mathtt{p1} \rangle\rangle}$
  $\mathrm{and}_{\mathrm{pred1}} =_{\mathrm{def}} (\lambda Y_{\mathtt{p1}} \,.\, (\lambda Z_{\mathtt{p1}} \,.\, (\lambda x_{\mathtt{e}} \,.\, (Yx \wedge Zx))))$
- $\mathrm{every}_{\mathrm{pred1}} \in \mathrm{Exp}_{\langle \mathtt{p1}, \langle \mathtt{p1}, \mathtt{t} \rangle\rangle}$
  $\mathrm{every}_{\mathrm{p1}} =_{\mathrm{def}} (\lambda Y_{\mathtt{p1}} \,.\, (\lambda Z_{\mathtt{p1}} \,.\, \forall x_{\mathtt{e}}(Yx \rightarrow Zx)))$

Lambda operator in the lambda calculus

- If $f \in \text{Exp}_\beta$ and $x \in \text{Var}_\alpha$, then $(\lambda x \,.\, f) \in \text{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\mathsf{e} \mid \mathsf{t} \mid \langle\alpha,\beta\rangle$
  $\mathsf{p1} =_{\text{def}} \langle \mathsf{e}, \mathsf{t} \rangle$
- $\text{and}_{\text{pred1}} \in \text{Exp}_{\langle \mathsf{p1}, \langle \mathsf{p1}, \mathsf{p1} \rangle\rangle}$
  $\text{and}_{\text{pred1}} =_{\text{def}} (\lambda Y_{\mathsf{p1}} \,.\, (\lambda Z_{\mathsf{p1}} \,.\, (\lambda x_{\mathsf{e}} \,.\, (Yx \wedge Zx))))$
- $\text{every}_{\text{pred1}} \in \text{Exp}_{\langle \mathsf{p1}, \langle \mathsf{p1}, \mathsf{t} \rangle\rangle}$
  $\text{every}_{\text{p1}} =_{\text{def}} (\lambda Y_{\mathsf{p1}} \,.\, (\lambda Z_{\mathsf{p1}} \,.\, \forall x_{\mathsf{e}}(Yx \to Zx)))$
- $\text{Mary} \in \text{Exp}_{\langle \mathsf{p1}, \mathsf{t} \rangle}$

Lambda operator in the lambda calculus

- If $f \in \mathrm{Exp}_\beta$ and $x \in \mathrm{Var}_\alpha$, then $(\lambda x\,.\,f) \in \mathrm{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\mathsf{e} \mid \mathsf{t} \mid \langle\alpha,\beta\rangle$
  $\mathsf{p1} =_{\mathsf{def}} \langle\mathsf{e},\mathsf{t}\rangle$
- $\mathsf{and}_{\mathsf{pred1}} \in \mathrm{Exp}_{\langle\mathsf{p1},\langle\mathsf{p1},\mathsf{p1}\rangle\rangle}$
  $\mathsf{and}_{\mathsf{pred1}} =_{\mathsf{def}} (\lambda Y_{\mathsf{p1}}\,.\,(\lambda Z_{\mathsf{p1}}\,.\,(\lambda x_{\mathsf{e}}\,.\,(Yx \wedge Zx))))$
- $\mathsf{every}_{\mathsf{pred1}} \in \mathrm{Exp}_{\langle\mathsf{p1},\langle\mathsf{p1},\mathsf{t}\rangle\rangle}$
  $\mathsf{every}_{\mathsf{p1}} =_{\mathsf{def}} (\lambda Y_{\mathsf{p1}}\,.\,(\lambda Z_{\mathsf{p1}}\,.\,\forall x_{\mathsf{e}}(Yx \to Zx)))$
- $\mathsf{Mary} \in \mathrm{Exp}_{\langle\mathsf{p1},\mathsf{t}\rangle}$
  $\mathsf{Mary} =_{\mathsf{def}} (\lambda Y_{\mathsf{p1}}\,.\,Y\mathsf{m}_{\mathsf{e}})$

Lambda operator in the lambda calculus

- If $f \in \text{Exp}_\beta$ and $x \in \text{Var}_\alpha$, then $(\lambda x\,.\,f) \in \text{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\mathsf{e} \mid \mathsf{t} \mid \langle\alpha,\beta\rangle$
  $\mathsf{p1} =_{\text{def}} \langle \mathsf{e}, \mathsf{t}\rangle$
- $\text{and}_{\text{pred1}} \in \text{Exp}_{\langle \mathsf{p1}, \langle \mathsf{p1}, \mathsf{p1}\rangle\rangle}$
  $\text{and}_{\text{pred1}} =_{\text{def}} (\lambda Y_{\mathsf{p1}}\,.\,(\lambda Z_{\mathsf{p1}}\,.\,(\lambda x_{\mathsf{e}}\,.\,(Yx \wedge Zx))))$
- $\text{every}_{\text{pred1}} \in \text{Exp}_{\langle \mathsf{p1}, \langle \mathsf{p1}, \mathsf{t}\rangle\rangle}$
  $\text{every}_{\text{p1}} =_{\text{def}} (\lambda Y_{\mathsf{p1}}\,.\,(\lambda Z_{\mathsf{p1}}\,.\,\forall x_{\mathsf{e}}(Yx \rightarrow Zx)))$
- $\text{Mary} \in \text{Exp}_{\langle \mathsf{p1}, \mathsf{t}\rangle}$
  $\text{Mary} =_{\text{def}} (\lambda Y_{\mathsf{p1}}\,.\,Y\text{m}_{\mathsf{e}})$
- $\text{not}_{\text{adj}} \in \text{Exp}_{\langle\langle \mathsf{p1}, \mathsf{p1}\rangle, \langle \mathsf{p1}, \mathsf{p1}\rangle\rangle}$

Lambda operator in the lambda calculus

- If $f \in \mathrm{Exp}_\beta$ and $x \in \mathrm{Var}_\alpha$, then $(\lambda x \,.\, f) \in \mathrm{Exp}_{\langle\alpha,\beta\rangle}$
- Types: $\mathsf{e} \mid \mathsf{t} \mid \langle\alpha,\beta\rangle$
  $\mathsf{p1} =_{\mathrm{def}} \langle \mathsf{e}, \mathsf{t}\rangle$
- $\mathrm{and}_{\mathrm{pred1}} \in \mathrm{Exp}_{\langle \mathsf{p1},\langle \mathsf{p1},\mathsf{p1}\rangle\rangle}$
  $\mathrm{and}_{\mathrm{pred1}} =_{\mathrm{def}} (\lambda Y_{\mathsf{p1}} \,.\, (\lambda Z_{\mathsf{p1}} \,.\, (\lambda x_{\mathsf{e}} \,.\, (Yx \wedge Zx))))$
- $\mathrm{every}_{\mathrm{pred1}} \in \mathrm{Exp}_{\langle \mathsf{p1},\langle \mathsf{p1},\mathsf{t}\rangle\rangle}$
  $\mathrm{every}_{\mathrm{p1}} =_{\mathrm{def}} (\lambda Y_{\mathsf{p1}} \,.\, (\lambda Z_{\mathsf{p1}} \,.\, \forall x_{\mathsf{e}}(Yx \to Zx)))$
- $\mathrm{Mary} \in \mathrm{Exp}_{\langle \mathsf{p1},\mathsf{t}\rangle}$
  $\mathrm{Mary} =_{\mathrm{def}} (\lambda Y_{\mathsf{p1}} \,.\, Y\mathrm{m}_{\mathsf{e}})$
- $\mathrm{not}_{\mathrm{adj}} \in \mathrm{Exp}_{\langle\langle \mathsf{p1},\mathsf{p1}\rangle,\langle \mathsf{p1},\mathsf{p1}\rangle\rangle}$
  $\mathrm{not}_{\mathrm{adj}} =_{\mathrm{def}} (\lambda W_{\langle \mathsf{p1},\mathsf{p1}\rangle} \,.\, (\lambda Y_{\mathsf{p1}} \,.\, (\lambda x_{\mathsf{e}} \,.\, \neg W(Yx))))$

Lambda operator in Haskell [1]

- This one works:
  `map (*2) [1,2,3]`

[1] Thanks, Matyi!

Lambda operator in Haskell [1]

- This one works:
  `map (*2) [1,2,3]`
- This one raises an error:
  `map (* 2 - 1) [1,2,3]`

[1] Thanks, Matyi!

Lambda operator in Haskell [1]

- This one works:
  `map (*2) [1,2,3]`
- This one raises an error:
  `map (* 2 - 1) [1,2,3]`
- Solution with `where`:
  ```
  map f [1,2,3] where
    f x = x * 2 - 1
  ```

[1] Thanks, Matyi!

Lambda operator in Haskell 1

- This one works:
  ```
  map (*2) [1,2,3]
  ```
- This one raises an error:
  ```
  map (* 2 - 1) [1,2,3]
  ```
- Solution with `where`:
  ```
  map f [1,2,3] where
    f x = x * 2 - 1
  ```
- Inline solution with composition:[1]
  ```
  map ((+(-1)) . (*)) [1,2,3]
  ```

[1] Thanks, Matyi!

Lambda operator in Haskell 1

- This one works:
  `map (*2) [1,2,3]`
- This one raises an error:
  `map (* 2 - 1) [1,2,3]`
- Solution with `where`:
  ```
  map f [1,2,3] where
    f x = x * 2 - 1
  ```
- Inline solution with composition:[1]
  `map ((+(-1)) . (*)) [1,2,3]`
- Inline solution with lambda:
  `map (\x -> x * 2 - 1) [1,2,3]`

[1] Thanks, Matyi!

Lambda operator in Haskell 2

- Lambda with multiple variables:
  ```
  foldr (\x y -> x * 2 - y) 1 [1,2,3]
  ```

Lambda operator in Haskell 2

- Lambda with multiple variables:
  ```
  foldr (\x y -> x * 2 - y) 1 [1,2,3]
  ```
- Creating constant function:
  ```
  \x -> 2
  ```

Lambda operator in Haskell 2

- Lambda with multiple variables:
  `foldr (\x y -> x * 2 - y) 1 [1,2,3]`
- Creating constant function:
  `\x -> 2`
- With pattern matching:
  uncurry' f = `\(y,x) -> f x y`

Lambda operator in Haskell 2

- Lambda with multiple variables:
  `foldr (\x y -> x * 2 - y) 1 [1,2,3]`
- Creating constant function:
  `\x -> 2`
- With pattern matching:
  uncurry' f = `\(y,x) -> f x y`
- Another example with pattern matching:
  `\(x:xs) -> xs ++ [x]`

Lambda operator in Haskell 2

- Lambda with multiple variables:
  `foldr (\x y -> x * 2 - y) 1 [1,2,3]`
- Creating constant function:
  `\x -> 2`
- With pattern matching:
  uncurry' f = `\(y,x) -> f x y`
- Another example with pattern matching:
  `\(x:xs) -> xs ++ [x]`
- Using with higher types:

```
filtermap ::   (a -> b) ->  (a -> Bool) ->  [a] ->  [b]
filtermap = \f p xs -> [f x | x <- xs, p x]
```

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' (1 : (2 : (3 : [])))
```

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
  reverse' (1 : (2 : (3 : [])))
1 reverse' (2 : (3 : [])) ++ 1 : []
```

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' (1 : (2 : (3 : [])))
```

1. `reverse' (2 : (3 : [])) ++ 1 : []`
2. `reverse' (3 : []) ++ 2 : [] ++ 1 : []`

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
  reverse' (1 : (2 : (3 : [])))
1 reverse' (2 : (3 : [])) ++ 1 : []
2 reverse' (3 : []) ++ 2 : [] ++ 1 : []
3 reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []
```

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
  reverse' (1 : (2 : (3 : [])))
1 reverse' (2 : (3 : [])) ++ 1 : []
2 reverse' (3 : []) ++ 2 : [] ++ 1 : []
3 reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []
4 [] ++ 3 : [] ++ 2 : [] ++ 1 : []
```

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' (1 : (2 : (3 : [])))
```

1. `reverse' (2 : (3 : [])) ++ 1 : []`
2. `reverse' (3 : []) ++ 2 : [] ++ 1 : []`
3. `reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []`
4. `[] ++ 3 : [] ++ 2 : [] ++ 1 : []`
5. `3 : [] ++ 2 : [] ++ 1 : []`

Reversing a list 1

An old friend:

```haskell
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```haskell
reverse' (1 : (2 : (3 : [])))
```

1. `reverse' (2 : (3 : [])) ++ 1 : []`
2. `reverse' (3 : []) ++ 2 : [] ++ 1 : []`
3. `reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []`
4. `[] ++ 3 : [] ++ 2 : [] ++ 1 : []`
5. `3 : [] ++ 2 : [] ++ 1 : []`
6. `3 : ([] ++ 2 : []) ++ 1 : []`

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
reverse' (1 : (2 : (3 : [])))
```

1. `reverse' (2 : (3 : [])) ++ 1 : []`
2. `reverse' (3 : []) ++ 2 : [] ++ 1 : []`
3. `reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []`
4. `[] ++ 3 : [] ++ 2 : [] ++ 1 : []`
5. `3 : [] ++ 2 : [] ++ 1 : []`
6. `3 : ([] ++ 2 : []) ++ 1 : []`
7. `3 : (2 : []) ++ 1 : []`

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
  reverse' (1 : (2 : (3 : [])))
1 reverse' (2 : (3 : [])) ++ 1 : []
2 reverse' (3 : []) ++ 2 : [] ++ 1 : []
3 reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []
4 [] ++ 3 : [] ++ 2 : [] ++ 1 : []
5 3 : [] ++ 2 : [] ++ 1 : []
6 3 : ([] ++ 2 : []) ++ 1 : []
7 3 : (2 : []) ++ 1 : []
8 3 : (2 : [] ++ 1 : [])
```

Reversing a list 1

An old friend:

```
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
  reverse' (1 : (2 : (3 : [])))
1 reverse' (2 : (3 : [])) ++ 1 : []
2 reverse' (3 : []) ++ 2 : [] ++ 1 : []
3 reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []
4 [] ++ 3 : [] ++ 2 : [] ++ 1 : []
5 3 : [] ++ 2 : [] ++ 1 : []
6 3 : ([] ++ 2 : []) ++ 1 : []
7 3 : (2 : []) ++ 1 : []
8 3 : (2 : [] ++ 1 : [])
9 3 : (2 : ([] ++ 1 : []))
```

Reversing a list 1

An old friend:

```haskell
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
   reverse' (1 : (2 : (3 : [])))
1  reverse' (2 : (3 : [])) ++ 1 : []
2  reverse' (3 : []) ++ 2 : [] ++ 1 : []
3  reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []
4  [] ++ 3 : [] ++ 2 : [] ++ 1 : []
5  3 : [] ++ 2 : [] ++ 1 : []
6  3 : ([] ++ 2 : []) ++ 1 : []
7  3 : (2 : []) ++ 1 : []
8  3 : (2 : [] ++ 1 : [])
9  3 : (2 : ([] ++ 1 : []))
10 3 : (2 : (1 : []))
```

Reversing a list 1

An old friend:

```haskell
reverse' :: [a] -> [a]
reverse' []     = []
reverse' (x:xs) = reverse' xs ++ [x]
```

```
   reverse' (1 : (2 : (3 : [])))
1  reverse' (2 : (3 : [])) ++ 1 : []
2  reverse' (3 : []) ++ 2 : [] ++ 1 : []
3  reverse' [] ++ 3 : [] ++ 2 : [] ++ 1 : []
4  [] ++ 3 : [] ++ 2 : [] ++ 1 : []
5  3 : [] ++ 2 : [] ++ 1 : []
6  3 : ([] ++ 2 : []) ++ 1 : []
7  3 : (2 : []) ++ 1 : []
8  3 : (2 : [] ++ 1 : [])
9  3 : (2 : ([] ++ 1 : []))
10 3 : (2 : (1 : []))
```

$1 + 2 + \cdots + (n+1) = \frac{(n+1)(n+2)}{2}$ recursive steps $\Longrightarrow$ quadratic time

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
```

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs

reverseA [] (1 : (2 : (3 : [])))
```

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

```
reverseA [] (1 : (2 : (3 : [])))
```

1. `reverseA (1 : []) (2 : (3 : []))`

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

```
reverseA [] (1 : (2 : (3 : [])))
```

1. `reverseA (1 : []) (2 : (3 : []))`
2. `reverseA (2 : (1 : [])) (3 : [])`

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

```
reverseA [] (1 : (2 : (3 : [])))
```

1. `reverseA (1 : []) (2 : (3 : []))`
2. `reverseA (2 : (1 : [])) (3 : [])`
3. `reverseA (3 : (2 : (1 : []))) []`

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

```
reverseA [] (1 : (2 : (3 : [])))
```

1. `reverseA (1 : []) (2 : (3 : []))`
2. `reverseA (2 : (1 : [])) (3 : [])`
3. `reverseA (3 : (2 : (1 : []))) []`
4. `(3 : (2 : (1 : [])))`

Reversing a list 2

An improved version:

```haskell
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

```haskell
  reverseA [] (1 : (2 : (3 : [])))
1 reverseA (1 : []) (2 : (3 : []))
2 reverseA (2 : (1 : [])) (3 : [])
3 reverseA (3 : (2 : (1 : []))) []
4 (3 : (2 : (1 : [])))
```

$n + 1$ recursive steps $\Longrightarrow$ linear time

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

```
reverseA [] (1 : (2 : (3 : [])))
```

1. `reverseA (1 : []) (2 : (3 : []))`
2. `reverseA (2 : (1 : [])) (3 : [])`
3. `reverseA (3 : (2 : (1 : []))) []`
4. `(3 : (2 : (1 : [])))`

$n + 1$ recursive steps $\Longrightarrow$ linear time: `reverse''` is $\frac{n+2}{2}$ times faster

Reversing a list 2

An improved version:

```
reverse'' :: [a] -> [a]
reverse'' xs = reverseA [] xs  where
  reverseA ys []     = ys
  reverseA ys (x:xs) = reverseA (x:ys) xs
```

```
reverseA [] (1 : (2 : (3 : [])))
```

1. `reverseA (1 : []) (2 : (3 : []))`
2. `reverseA (2 : (1 : [])) (3 : [])`
3. `reverseA (3 : (2 : (1 : []))) []`
4. `(3 : (2 : (1 : [])))`

$n+1$ recursive steps $\Longrightarrow$ linear time: `reverse''` is $\frac{n+2}{2}$ times faster

But: does it do the same?

Equivalence

Equivalence Let `b` be an instance of `Eq`. Now

$$\text{‘}\texttt{f1 :: a -> b}\text{’} \approx \text{‘}\texttt{f2 :: a -> b}\text{’}$$

iff

Equivalence

Equivalence Let `b` be an instance of `Eq`. Now

$$\text{‘}\texttt{f1 :: a -> b}\text{’} \approx \text{‘}\texttt{f2 :: a -> b}\text{’}$$

iff

1. ‘`f1`’ returns a value iff ‘`f2`’ does;

Equivalence

Equivalence Let `b` be an instance of `Eq`. Now

‘`f1 :: a -> b`’ $\approx$ ‘`f2 :: a -> b`’

iff

1. ‘`f1`’ returns a value iff ‘`f2`’ does;
2. ‘`f1 x == f2 x`’ returns ‘`True`’ whenever there’s a return value

Equivalence

Equivalence Let `b` be an instance of `Eq`. Now

$$`\texttt{f1 :: a -> b}' \approx `\texttt{f2 :: a -> b}'$$

iff

1. ‘`f1`’ returns a value iff ‘`f2`’ does;
2. ‘`f1 x == f2 x`’ returns ‘`True`’ whenever there's a return value

$$`\texttt{t1 :: b}' \approx_{\texttt{x :: a}} `\texttt{t2 :: b}'$$

iff

Equivalence

Equivalence Let `b` be an instance of `Eq`. Now

$$\text{`}\texttt{f1 :: a -> b}\text{'} \approx \text{`}\texttt{f2 :: a -> b}\text{'}$$

iff

1. ‘`f1`’ returns a value iff ‘`f2`’ does;
2. ‘`f1 x == f2 x`’ returns ‘`True`’ whenever there's a return value

$$\text{`}\texttt{t1 :: b}\text{'} \approx_{\texttt{x :: a}} \text{`}\texttt{t2 :: b}\text{'}$$

iff

$$\text{`}\texttt{\textbackslash x -> t1 :: a -> b}\text{'} \approx \text{`}\texttt{\textbackslash x -> t2 :: a -> b}\text{'}$$

Proving equivalence by induction

Theorem ‘`reverse'`’ $\approx$ ‘`reverse''`’

Proving equivalence by induction

Theorem ‘`reverse'`’ $\approx$ ‘`reverse''`’

Proof Attempt 1: induction on the complexity of `xs`.

Proving equivalence by induction

Theorem ‘`reverse'`’ ≈ ‘`reverse''`’

Proof Attempt 1: induction on the complexity of `xs`.

Base: ‘`reverse' [] == reverse'' []`’ returns ‘`True`’ by definition.

Proving equivalence by induction

Theorem ‘`reverse'`’ ≈ ‘`reverse''`’

Proof Attempt 1: induction on the complexity of `xs`.

Base: ‘`reverse' [] == reverse'' []`’ returns ‘`True`’ by definition.

Hypothesis: ‘`reverse' xs == reverse'' xs`’ returns ‘`True`’

Proving equivalence by induction

Theorem ‘`reverse'`’ $\approx$ ‘`reverse''`’

Proof Attempt 1: induction on the complexity of `xs`.

Base: ‘`reverse' [] == reverse'' []`’
returns ‘`True`’ by definition.

Hypothesis: ‘`reverse' xs == reverse'' xs`’
returns ‘`True`’

Ind. step: `reverse' (x:xs)`

Proving equivalence by induction

Theorem ‘`reverse'`’ ≈ ‘`reverse''`’

Proof Attempt 1: induction on the complexity of `xs`.

Base: ‘`reverse' [] == reverse'' []`’
returns ‘`True`’ by definition.

Hypothesis: ‘`reverse' xs == reverse'' xs`’
returns ‘`True`’

Ind. step: `reverse' (x:xs)`
`(reverse' xs) ++ [x]` by def.

Proving equivalence by induction

Theorem ‘`reverse'`’ ≈ ‘`reverse''`’

Proof Attempt 1: induction on the complexity of `xs`.

Base: ‘`reverse' [] == reverse'' []`’ returns ‘`True`’ by definition.

Hypothesis: ‘`reverse' xs == reverse'' xs`’ returns ‘`True`’

Ind. step: `reverse' (x:xs)`
`(reverse' xs) ++ [x]` by def.
`(reverse'' xs) ++ [x]` by hyp.

Proving equivalence by induction

Theorem `'reverse'` $\approx$ `'reverse''`

Proof Attempt 1: induction on the complexity of `xs`.

Base: `'reverse' [] == reverse'' []'`
returns 'True' by definition.

Hypothesis: `'reverse' xs == reverse'' xs'`
returns 'True'

Ind. step: `reverse' (x:xs)`
`(reverse' xs) ++ [x]` by def.
`(reverse'' xs) ++ [x]` by hyp.
`(reverseA [] xs) ++ [x]` by def.

Proving equivalence by induction

Theorem ‘`reverse'`’ ≈ ‘`reverse''`’

Proof Attempt 1: induction on the complexity of `xs`.

Base: ‘`reverse' [] == reverse'' []`’ returns ‘`True`’ by definition.

Hypothesis: ‘`reverse' xs == reverse'' xs`’ returns ‘`True`’

Ind. step: `reverse' (x:xs)`
`(reverse' xs) ++ [x]` by def.
`(reverse'' xs) ++ [x]` by hyp.
`(reverseA [] xs) ++ [x]` by def.
No inductive hypothesis on `reverseA`...

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

Base: ‘`reverse' [] ++ ys == reverseA ys []`’ returns ‘`True`’ by def.

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

Base: ‘`reverse' [] ++ ys == reverseA ys []`’ returns ‘`True`’ by def.

Hypothesis: ‘`reverse' xs ++ ys`’ $\approx_{\texttt{ys}}$ ‘`reverseA ys xs`’

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

- **Base:** ‘`reverse' [] ++ ys == reverseA ys []`’ returns ‘`True`’ by def.
- **Hypothesis:** ‘`reverse' xs ++ ys`’ $\approx_{\texttt{ys}}$ ‘`reverseA ys xs`’
- **Ind. step:** `reverse' (x:xs) ++ ys`

Proving equivalence by induction 2

Lemma 'reverse' xs ++ ys' $\approx_{\texttt{xs}}$ 'reverseA ys xs'

Proof Attempt 2: induction on the complexity of xs.

Base: 'reverse' [] ++ ys == reverseA ys []'
returns 'True' by def.

Hypothesis: 'reverse' xs ++ ys' $\approx_{\texttt{ys}}$ 'reverseA ys xs'

Ind. step: reverse' (x:xs) ++ ys
(reverse' xs) ++ [x] ++ ys by def.

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

Base: ‘`reverse' [] ++ ys == reverseA ys []`’ returns ‘`True`’ by def.

Hypothesis: ‘`reverse' xs ++ ys`’ $\approx_{\texttt{ys}}$ ‘`reverseA ys xs`’

Ind. step: `reverse' (x:xs) ++ ys`

`(reverse' xs) ++ [x] ++ ys` by def.

`(reverse'' xs) ++ x:ys` by def. of ‘++‘

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

Base: ‘`reverse' [] ++ ys == reverseA ys []`’
returns ‘`True`’ by def.

Hypothesis: ‘`reverse' xs ++ ys`’ $\approx_{\texttt{ys}}$ ‘`reverseA ys xs`’

Ind. step: `reverse' (x:xs) ++ ys`
`(reverse' xs) ++ [x] ++ ys` by def.
`(reverse'' xs) ++ x:ys` by def. of ‘++‘
`(reverseA (x:ys) xs` by hyp.

Proving equivalence by induction 2

Lemma ‘`reverse' xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

Base: ‘`reverse' [] ++ ys == reverseA ys []`’ returns ‘`True`’ by def.

Hypothesis: ‘`reverse' xs ++ ys`’ $\approx_{\texttt{ys}}$ ‘`reverseA ys xs`’

Ind. step: `reverse' (x:xs) ++ ys`

`(reverse' xs) ++ [x] ++ ys` by def.

`(reverse'' xs) ++ x:ys` by def. of ‘++‘

`(reverseA (x:ys) xs` by hyp.

`(reverseA ys (x:xs)` by def.

Proving equivalence by induction 2

Lemma ‘`reverse'` `xs ++ ys`’ $\approx_{\texttt{xs}}$ ‘`reverseA ys xs`’

Proof Attempt 2: induction on the complexity of `xs`.

Base: ‘`reverse' [] ++ ys == reverseA ys []`’
returns ‘`True`’ by def.

Hypothesis: ‘`reverse' xs ++ ys`’ $\approx_{\texttt{ys}}$ ‘`reverseA ys xs`’

Ind. step: `reverse' (x:xs) ++ ys`
`(reverse' xs) ++ [x] ++ ys` by def.
`(reverse'' xs) ++ x:ys` by def. of ‘++‘
`(reverseA (x:ys) xs` by hyp.
`(reverseA ys (x:xs)` by def.
Q. E. D.