

# Functional Programming and Verification

## revision course

Jonas Hübötter

August 19, 2020

# Outline

Functional Programming and Haskell

Types

Proofs

Correctness

I/O

Evaluation

Time complexity analysis

# Plan

## Functional Programming and Haskell

- Basic Haskell

- Recursion, guards, pattern matching

- List comprehensions

- QuickCheck

- Polymorphism

- Currying, partial application, higher-order functions

# Basic Haskell

function types

$f :: a \rightarrow b \rightarrow c$

# Basic Haskell

function types

$f :: a \rightarrow b \rightarrow c$

function definitions

$f\ x\ y = \dots$

# Basic Haskell

function types	<code>f :: a -&gt; b -&gt; c</code>
function definitions	<code>f x y = ...</code>
function application	<code>f 1 2</code>

# Basic Haskell

function types	<code>f :: a -&gt; b -&gt; c</code>
function definitions	<code>f x y = ...</code>
function application	<code>f 1 2</code>
conditional	<code>if True then a else b</code>

# Basic Haskell

function types	<code>f :: a -&gt; b -&gt; c</code>
function definitions	<code>f x y = ...</code>
function application	<code>f 1 2</code>
conditional	<code>if True then a else b</code>
prefix/infix precedence	<code>f a 'g' b</code> means <code>(f a) 'g' b</code>



# Basic Haskell

function types	<code>f :: a -&gt; b -&gt; c</code>
function definitions	<code>f x y = ...</code>
function application	<code>f 1 2</code>
conditional	<code>if True then a else b</code>
prefix/infix precedence	<code>f a 'g' b</code> means <code>(f a) 'g' b</code>
\$ sign	<code>f \$ a 'g' b</code> means <code>f (a 'g' b)</code>

# Types

Bool

True or False

# Types

Bool

True or False

Int

fixed-width integers

# Types

Bool	True or False
Int	fixed-width integers
Integer	unbounded integers

# Types

Bool	True or False
Int	fixed-width integers
Integer	unbounded integers
Char	'a'

# Types

Bool	True or False
Int	fixed-width integers
Integer	unbounded integers
Char	'a'
String	"hello" :: [Char]

# Types

Bool	True or False
Int	fixed-width integers
Integer	unbounded integers
Char	'a'
String	"hello" :: [Char]
(a,b) (Tuple)	("hello",1) :: (String,Int)

# Tuples

```
(1,"hello") :: (Int,String)
(x,y,z)    :: (a,b,c)
-- ...
```

Prelude functions: `fst`, `snd`



# Lists

Two ways of constructing a list:

# Lists

Two ways of constructing a list:

```
a = [1,2,3]           :: [Int]
```

# Lists

Two ways of constructing a list:

```
a = [1,2,3]           :: [Int]
```

```
b = 1 : 2 : 3 : []    :: [Int]
```

# Lists

Two ways of constructing a list:

```
a = [1,2,3]           :: [Int]
```

```
b = 1 : 2 : 3 : []    :: [Int]
```

Cons (:) and [] are **constructors** of lists, that is a function that **uniquely constructs** a value of the list type.

# Lists

Two ways of constructing a list:

```
a = [1,2,3]           :: [Int]
```

```
b = 1 : 2 : 3 : []    :: [Int]
```

Cons (:) and [] are **constructors** of lists, that is a function that **uniquely constructs** a value of the list type.

Intuitively:  $(:) :: a \rightarrow [a] \rightarrow [a]$ .

## Prelude functions

`head :: [a] -> a`

first element

## Prelude functions

`head :: [a] -> a`

`last :: [a] -> a`

first element

last element

## Prelude functions

`head :: [a] -> a`

`last :: [a] -> a`

`init :: [a] -> [a]`

first element

last element

every element but last  
element



## Prelude functions

`head :: [a] -> a`

`last :: [a] -> a`

`init :: [a] -> [a]`

`tail :: [a] -> [a]`

first element

last element

every element but last  
element

every element but first  
element

## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

`(++) :: [a] -> [a] -> [a]`

append lists

## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

`(++) :: [a] -> [a] -> [a]`

append lists

`reverse :: [a] -> [a]`

reverse list

## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

`(++) :: [a] -> [a] -> [a]`

append lists

`reverse :: [a] -> [a]`

reverse list

`length :: [a] -> Int`

length of list

## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

`(++) :: [a] -> [a] -> [a]`

append lists

`reverse :: [a] -> [a]`

reverse list

`length :: [a] -> Int`

length of list

`null :: [a] -> Bool`

empty?

## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

`(++) :: [a] -> [a] -> [a]`

append lists

`reverse :: [a] -> [a]`

reverse list

`length :: [a] -> Int`

length of list

`null :: [a] -> Bool`

empty?

`concat :: [[a]] -> [a]`

flatten list

## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

`(++) :: [a] -> [a] -> [a]`

append lists

`reverse :: [a] -> [a]`

reverse list

`length :: [a] -> Int`

length of list

`null :: [a] -> Bool`

empty?

`concat :: [[a]] -> [a]`

flatten list

`zip :: [a] -> [b] -> [(a,b)]`

combine lists element-wise



## Prelude functions

`head :: [a] -> a`

first element

`last :: [a] -> a`

last element

`init :: [a] -> [a]`

every element but last  
element

`tail :: [a] -> [a]`

every element but first  
element

`elem :: a -> [a] -> Bool`

element in list?

`(++) :: [a] -> [a] -> [a]`

append lists

`reverse :: [a] -> [a]`

reverse list

`length :: [a] -> Int`

length of list

`null :: [a] -> Bool`

empty?

`concat :: [[a]] -> [a]`

flatten list

`zip :: [a] -> [b] -> [(a,b)]`

combine lists element-wise

`unzip :: [(a,b)] -> ([a],[b])`

separate list of tuples into  
list of components

## Prelude functions (2)

`replicate :: Int -> a -> [a]`    build list from repeated element

## Prelude functions (2)

<code>replicate :: Int -&gt; a -&gt; [a]</code>	build list from repeated element
<code>take :: Int -&gt; [a] -&gt; [a]</code>	prefix of list with given length

## Prelude functions (2)

<code>replicate :: Int -&gt; a -&gt; [a]</code>	build list from repeated element
<code>take :: Int -&gt; [a] -&gt; [a]</code>	prefix of list with given length
<code>drop :: Int -&gt; [a] -&gt; [a]</code>	list without prefix with given length

## Prelude functions (2)

<code>replicate :: Int -&gt; a -&gt; [a]</code>	build list from repeated element
<code>take :: Int -&gt; [a] -&gt; [a]</code>	prefix of list with given length
<code>drop :: Int -&gt; [a] -&gt; [a]</code>	list without prefix with given length
<code>and :: [Bool] -&gt; Bool</code>	conjunction over all elements

## Prelude functions (2)

<code>replicate :: Int -&gt; a -&gt; [a]</code>	build list from repeated element
<code>take :: Int -&gt; [a] -&gt; [a]</code>	prefix of list with given length
<code>drop :: Int -&gt; [a] -&gt; [a]</code>	list without prefix with given length
<code>and :: [Bool] -&gt; Bool</code>	conjunction over all elements
<code>or :: [Bool] -&gt; Bool</code>	disjunction over all elements

## Prelude functions (2)

<code>replicate :: Int -&gt; a -&gt; [a]</code>	build list from repeated element
<code>take :: Int -&gt; [a] -&gt; [a]</code>	prefix of list with given length
<code>drop :: Int -&gt; [a] -&gt; [a]</code>	list without prefix with given length
<code>and :: [Bool] -&gt; Bool</code>	conjunction over all elements
<code>or :: [Bool] -&gt; Bool</code>	disjunction over all elements
<code>sum :: [Int] -&gt; Int</code>	sum over all elements

## Prelude functions (2)

<code>replicate :: Int -&gt; a -&gt; [a]</code>	build list from repeated element
<code>take :: Int -&gt; [a] -&gt; [a]</code>	prefix of list with given length
<code>drop :: Int -&gt; [a] -&gt; [a]</code>	list without prefix with given length
<code>and :: [Bool] -&gt; Bool</code>	conjunction over all elements
<code>or :: [Bool] -&gt; Bool</code>	disjunction over all elements
<code>sum :: [Int] -&gt; Int</code>	sum over all elements
<code>product :: [Int] -&gt; Int</code>	product over all elements



## Prelude functions (2)

`replicate :: Int -> a -> [a]`

build list from repeated  
element

`take :: Int -> [a] -> [a]`

prefix of list with given length

`drop :: Int -> [a] -> [a]`

list without prefix with given  
length

`and :: [Bool] -> Bool`

conjunction over all elements

`or :: [Bool] -> Bool`

disjunction over all elements

`sum :: [Int] -> Int`

sum over all elements

`product :: [Int] -> Int`

product over all elements

`(!!) :: [a] -> Int -> a`

get element at index

## Prelude functions (2)

<code>replicate :: Int -&gt; a -&gt; [a]</code>	build list from repeated element
<code>take :: Int -&gt; [a] -&gt; [a]</code>	prefix of list with given length
<code>drop :: Int -&gt; [a] -&gt; [a]</code>	list without prefix with given length
<code>and :: [Bool] -&gt; Bool</code>	conjunction over all elements
<code>or :: [Bool] -&gt; Bool</code>	disjunction over all elements
<code>sum :: [Int] -&gt; Int</code>	sum over all elements
<code>product :: [Int] -&gt; Int</code>	product over all elements
<code>(!!) :: [a] -&gt; Int -&gt; a</code>	get element at index

search for functions by type signature on  
<https://hoogle.haskell.org/>.

# Ranges

[1..5]

# Ranges

```
[1..5]  
= [1,2,3,4,5]
```

```
[1,3..10]
```

# Ranges

```
[1..5]  
= [1,2,3,4,5]
```

```
[1,3..10]  
= [1,3,5,7,9]
```

```
[1..]
```

# Ranges

`[1..5]`  
`= [1,2,3,4,5]`

`[1,3..10]`  
`= [1,3,5,7,9]`

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

`[1,3..]`

# Ranges

`[1..5]`  
`= [1,2,3,4,5]`

`[1,3..10]`  
`= [1,3,5,7,9]`

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

`[1,3..]`  
`= [1,3,5...]`

## Local definitions

`let x = e1 in e2`

defines  $x$  locally in  $e_2$ .



## Local definitions

`let x = e1 in e2`

defines  $x$  locally in  $e_2$ .

`e2 where x = e1`

also defines  $x$  locally in  $e_2$  where  $e_2$  has to be a function definition.

# Recursion, guards, pattern matching

## Guards

# Recursion, guards, pattern matching

## Guards

Example: maximum of two integers.

```
max2 :: Integer -> Integer -> Integer  
max2 x y
```

# Recursion, guards, pattern matching

## Guards

Example: maximum of two integers.

```
max2 :: Integer -> Integer -> Integer
max2 x y
  | x >= y    = x
  | otherwise = y
```

# Recursion

Reduce problem into a solving a series of smaller problems of a similar kind.

## Example

```
factorial :: Integer -> Integer
factorial n
```

# Recursion

Reduce problem into a solving a series of smaller problems of a similar kind.

## Example

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1           -- base case
  | n > 0  = n * factorial (n - 1)  -- recursive case
```

## Accumulating parameter

Alternatively, factorial could be defined as

```
factorial :: Integer -> Integer
```

## Accumulating parameter

Alternatively, factorial could be defined as

```
factorial :: Integer -> Integer
```

```
factorial n = aux n 1
```

```
  where
```

```
    aux :: Integer -> Integer -> Integer
```

```
    aux n acc
```



## Accumulating parameter

Alternatively, factorial could be defined as

```
factorial :: Integer -> Integer
```

```
factorial n = aux n 1
```

```
  where
```

```
    aux :: Integer -> Integer -> Integer
```

```
    aux n acc
```

```
      | n == 0 = acc
```

```
      | n > 0  = aux (n - 1) (n * acc)
```

## Accumulating parameter

Alternatively, factorial could be defined as

```
factorial :: Integer -> Integer
factorial n = aux n 1
  where
    aux :: Integer -> Integer -> Integer
    aux n acc
      | n == 0 = acc
      | n > 0  = aux (n - 1) (n * acc)
```

The resulting function is **tail recursive**, that is the recursive call is located at the very end of its body.

Therefore, no computation is done after the recursive function call returns.

## Accumulating parameter

Alternatively, factorial could be defined as

```
factorial :: Integer -> Integer
factorial n = aux n 1
  where
    aux :: Integer -> Integer -> Integer
    aux n acc
      | n == 0 = acc
      | n > 0  = aux (n - 1) (n * acc)
```

The resulting function is **tail recursive**, that is the recursive call is located at the very end of its body.

Therefore, no computation is done after the recursive function call returns.

In general, recursion using accumulating parameters is less readable.

## Pattern matching

A more compact syntax for recursion:

```
factorial 0 = 1
```

```
factorial n | n > 0 = n * factorial (n - 1)
```

# Pattern matching

A more compact syntax for recursion:

```
factorial 0 = 1  
factorial n | n > 0 = n * factorial (n - 1)
```

Patterns are expressions consisting only of constructors, variables, and literals.

# Pattern matching

## Examples

```
head :: [a] -> a
```

# Pattern matching

## Examples

```
head :: [a] -> a
```

```
head (x : _) = x
```

```
tail :: [a] -> [a]
```

# Pattern matching

## Examples

```
head :: [a] -> a
```

```
head (x : _) = x
```

```
tail :: [a] -> [a]
```

```
tail (_ : xs) = xs
```

```
null :: [a] -> Bool
```



# Pattern matching

## Examples

```
head :: [a] -> a  
head (x : _) = x
```

```
tail :: [a] -> [a]  
tail (_ : xs) = xs
```

```
null :: [a] -> Bool  
null []      = True  
null (_ : _) = False
```

# Constructors vs Types

What is the difference between `True` and `Bool`?

- `True` is a **constructor**, `Bool` is a **type**.
- `True` can be used **in expressions** to build values of a type.
- `Bool` can be used **in type signatures** to hint at the type of bindings.

# Constructors vs Types

What is the difference between True and Bool?

- True is a **constructor**, Bool is a **type**.
- True can be used **in expressions** to build values of a type.
- Bool can be used **in type signatures** to hint at the type of bindings.

Constructor?

- False

# Constructors vs Types

What is the difference between `True` and `Bool`?

- `True` is a **constructor**, `Bool` is a **type**.
- `True` can be used **in expressions** to build values of a type.
- `Bool` can be used **in type signatures** to hint at the type of bindings.

Constructor?

- `False`      **yes**
- `(:)`

# Constructors vs Types

What is the difference between True and Bool?

- True is a **constructor**, Bool is a **type**.
- True can be used **in expressions** to build values of a type.
- Bool can be used **in type signatures** to hint at the type of bindings.

Constructor?

- False      **yes**
- (:)        **yes**
- Maybe

# Constructors vs Types

What is the difference between `True` and `Bool`?

- `True` is a **constructor**, `Bool` is a **type**.
- `True` can be used **in expressions** to build values of a type.
- `Bool` can be used **in type signatures** to hint at the type of bindings.

Constructor?

- `False`      **yes**
- `(:)`        **yes**
- `Maybe`     **no**
- `Just`

# Constructors vs Types

What is the difference between `True` and `Bool`?

- `True` is a **constructor**, `Bool` is a **type**.
- `True` can be used **in expressions** to build values of a type.
- `Bool` can be used **in type signatures** to hint at the type of bindings.

Constructor?

- `False`      **yes**
- `(:)`        **yes**
- `Maybe`     **no**
- `Just`       **yes**
- `Nothing`

# Constructors vs Types

What is the difference between `True` and `Bool`?

- `True` is a **constructor**, `Bool` is a **type**.
- `True` can be used **in expressions** to build values of a type.
- `Bool` can be used **in type signatures** to hint at the type of bindings.

Constructor?

- `False`      **yes**
- `(:)`        **yes**
- `Maybe`     **no**
- `Just`       **yes**
- `Nothing`    **yes**



## Case

Pattern matching in nested expressions

```
singleOrEmpty :: [a] -> Bool
singleOrEmpty xs = case xs of []    -> True
                               [_]   -> True
                               _     -> False
```

## List comprehensions

$$[ \textit{expr} \mid E_1, \dots, E_n ]$$

where *expr* is an expression and each  $E_i$  is a generator or a test.

# List comprehensions

$$[ \textit{expr} \mid E_1, \dots, E_n ]$$

where *expr* is an expression and each  $E_i$  is a generator or a test.

- a **generator** is of the form *pattern* <- *listexpression*

# List comprehensions

$$[ \textit{expr} \mid E_1, \dots, E_n ]$$

where *expr* is an expression and each  $E_i$  is a generator or a test.

- a **generator** is of the form *pattern*  $\leftarrow$  *listexpression*
- a **test** is a Boolean expression

# List comprehensions

## Examples

```
[x ^ 2 | x <- [1..5]]
```

# List comprehensions

## Examples

```
[x ^ 2 | x <- [1..5]]  
= [1, 4, 9, 16, 25]
```

```
[toLower c | c <- "Hello World!"]
```

# List comprehensions

## Examples

```
[x ^ 2 | x <- [1..5]]  
= [1, 4, 9, 16, 25]
```

```
[toLower c | c <- "Hello World!"]  
= "hello world!"
```

```
[(x, even x) | x <- [1..3]]
```

# List comprehensions

## Examples

```
[x ^ 2 | x <- [1..5]]  
= [1, 4, 9, 16, 25]
```

```
[toLower c | c <- "Hello World!"]  
= "hello world!"
```

```
[(x, even x) | x <- [1..3]]  
= [(1, False), (2, True), (3, False)]
```



## Multiple generators

Generators are reduced from left to right.

A generator or test can depend on any generator to its left.

## Multiple generators

Generators are reduced from left to right.

A generator or test can depend on any generator to its left.

### Example

```
[(i,j) | i <- [1 .. 3], j <- [i .. 3]]
```

## Multiple generators

Generators are reduced from left to right.

A generator or test can depend on any generator to its left.

### Example

```
[(i,j) | i <- [1 .. 3], j <- [i .. 3]]  
= [(1,j) | j <- [1..3]] ++  
  [(2,j) | j <- [2..3]] ++  
  [(3,j) | j <- [3..3]]
```

## Multiple generators

Generators are reduced from left to right.

A generator or test can depend on any generator to its left.

### Example

```
[(i,j) | i <- [1 .. 3], j <- [i .. 3]]  
= [(1,j) | j <- [1..3]] ++  
  [(2,j) | j <- [2..3]] ++  
  [(3,j) | j <- [3..3]]  
= [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]
```

## The meaning of list comprehensions

`[e | x <- [a1,...,an]]`

## The meaning of list comprehensions

$$[e \mid x \leftarrow [a_1, \dots, a_n]]$$
$$= (\text{let } x = a_1 \text{ in } [e]) ++ \dots ++ (\text{let } x = a_n \text{ in } [e])$$
$$[e \mid b]$$

## The meaning of list comprehensions

$[e \mid x \leftarrow [a_1, \dots, a_n]]$   
 $= (\text{let } x = a_1 \text{ in } [e]) ++ \dots ++ (\text{let } x = a_n \text{ in } [e])$

$[e \mid b]$   
 $= \text{if } b \text{ then } [e] \text{ else } []$

$[e \mid x \leftarrow [a_1, \dots, a_n], E]$

## The meaning of list comprehensions

```
[e | x <- [a1,...,an]]  
= (let x = a1 in [e]) ++ . . . ++ (let x = an in [e])
```

```
[e | b]  
= if b then [e] else []
```

```
[e | x <- [a1,...,an], E]  
= (let x = a1 in [e | E]) ++ . . . ++  
  (let x = an in [e | E])
```

```
[e | b, E]
```



## The meaning of list comprehensions

```
[e | x <- [a1,...,an]]  
= (let x = a1 in [e]) ++ . . . ++ (let x = an in [e])
```

```
[e | b]  
= if b then [e] else []
```

```
[e | x <- [a1,...,an], E]  
= (let x = a1 in [e | E]) ++ . . . ++  
  (let x = an in [e | E])
```

```
[e | b, E]  
= if b then [e | E] else []
```

## QuickCheck

QuickCheck tests check if a proposition holds true for a large number of random arguments.

## QuickCheck

QuickCheck tests check if a proposition holds true for a large number of random arguments.

It can be used to *test* the equivalence of two functions.

## QuickCheck

QuickCheck tests check if a proposition holds true for a large number of random arguments.

It can be used to *test* the equivalence of two functions.

### Examples

```
import Test.QuickCheck
```

```
prop_max2 x y =
```

## QuickCheck

QuickCheck tests check if a proposition holds true for a large number of random arguments.

It can be used to *test* the equivalence of two functions.

### Examples

```
import Test.QuickCheck
```

```
prop_max2 x y =  
    max2 x y == max x y
```

```
prop_max2_assoc x y z =
```

## QuickCheck

QuickCheck tests check if a proposition holds true for a large number of random arguments.

It can be used to *test* the equivalence of two functions.

### Examples

```
import Test.QuickCheck
```

```
prop_max2 x y =  
    max2 x y == max x y
```

```
prop_max2_assoc x y z =  
    max2 x (max2 y z) == max2 (max2 x y) z
```

```
prop_factorial n =
```

## QuickCheck

QuickCheck tests check if a proposition holds true for a large number of random arguments.

It can be used to *test* the equivalence of two functions.

### Examples

```
import Test.QuickCheck
```

```
prop_max2 x y =  
    max2 x y == max x y
```

```
prop_max2_assoc x y z =  
    max2 x (max2 y z) == max2 (max2 x y) z
```

```
prop_factorial n =  
    n >= 0 ==> n < factorial n
```

## QuickCheck

QuickCheck tests check if a proposition holds true for a large number of random arguments.

It can be used to *test* the equivalence of two functions.

### Examples

```
import Test.QuickCheck
```

```
prop_max2 x y =  
    max2 x y == max x y
```

```
prop_max2_assoc x y z =  
    max2 x (max2 y z) == max2 (max2 x y) z
```

```
prop_factorial n =  
    n >= 0 ==> n < factorial n
```

Run `quickCheck prop_max2` from GHCI to check the property.



# Polymorphism

One function definition, having many types.

# Polymorphism

One function definition, having many types.

`length :: [a] -> Int` is defined for all types `a`  
where `a` is a **type variable**.

# Subtype vs parametric polymorphism

- **parametric polymorphism**  
types may contain universally quantified type variables that are then replaced by actual types.
- **subtype polymorphism**  
any object of type  $T'$  where  $T'$  is a subtype of  $T$  can be used in place of objects of type  $T$ .

# Subtype vs parametric polymorphism

- **parametric polymorphism**  
types may contain universally quantified type variables that are then replaced by actual types.
- **subtype polymorphism**  
any object of type  $T'$  where  $T'$  is a subtype of  $T$  can be used in place of objects of type  $T$ .

Haskell uses parametric polymorphism.

## Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** **Num**.

## Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** `Num`.

Some type classes:

## Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** `Num`.

Some type classes:

- `Num`

## Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** `Num`.

Some type classes:

- `Num`
- `Integral`



## Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** `Num`.

Some type classes:

- `Num`
- `Integral`
- `Fractional`

# Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** `Num`.

Some type classes:

- `Num`
- `Integral`
- `Fractional`
- `Ord`

# Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** **Num**.

Some type classes:

- **Num**
- **Integral**
- **Fractional**
- **Ord**
- **Eq**

# Type constraints

Type variables can be constrained by **type constraints**.

$(+)$  :: **Num** **a** =>  $a \rightarrow a \rightarrow a$

Function  $(+)$  has type  $a \rightarrow a \rightarrow a$  for any type  $a$  of the **type class** **Num**.

Some type classes:

- **Num**
- **Integral**
- **Fractional**
- **Ord**
- **Eq**
- **Show**

## Quiz

`f x y z = if x then y else z`

## Quiz

```
f x y z = if x then y else z
```

```
f :: Bool -> a -> a -> a
```

```
f x y = [(x,y), (y,x)]
```

## Quiz

```
f x y z = if x then y else z  
f :: Bool -> a -> a -> a
```

```
f x y = [(x,y), (y,x)]  
f :: a -> a -> [(a,a)]
```

```
f x = [length u + v | (u,v) <- x ]
```

## Quiz

```
f x y z = if x then y else z  
f :: Bool -> a -> a -> a
```

```
f x y = [(x,y), (y,x)]  
f :: a -> a -> [(a,a)]
```

```
f x = [length u + v | (u,v) <- x ]  
f :: [[a],Int)] -> [Int]
```

```
f x y = [u ++ x | u <- y, length u < x ]
```



## Quiz

```
f x y z = if x then y else z  
f :: Bool -> a -> a -> a
```

```
f x y = [(x,y), (y,x)]  
f :: a -> a -> [(a,a)]
```

```
f x = [length u + v | (u,v) <- x ]  
f :: [(a,Int)] -> [Int]
```

```
f x y = [u ++ x | u <- y, length u < x ]  
invalid
```

```
f x y = [(u,v) | u <- w, u, v <- x] | w <- y]
```

## Quiz

```
f x y z = if x then y else z  
f :: Bool -> a -> a -> a
```

```
f x y = [(x,y), (y,x)]  
f :: a -> a -> [(a,a)]
```

```
f x = [length u + v | (u,v) <- x ]  
f :: [(a,Int)] -> [Int]
```

```
f x y = [u ++ x | u <- y, length u < x ]  
invalid
```

```
f x y = [(u,v) | u <- w, u, v <- x] | w <- y]  
f :: [a] -> [[Bool]] -> [(Bool, a)]
```

# Currying

A function is **curried** when it takes its arguments one at a time, each time returning a new function.

# Currying

A function is **curried** when it takes its arguments one at a time, each time returning a new function.

## Example

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$	$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
$f\ x\ y = x + y$	$f\ x = \backslash y \rightarrow x + y$

# Currying

A function is **curried** when it takes its arguments one at a time, each time returning a new function.

## Example

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$	$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
---	---

$f\ x\ y = x + y$	$f\ x = \backslash y \rightarrow x + y$
-------------------	---

$f\ a\ b$	$(f\ a)\ b$
-----------	-------------

# Currying

A function is **curried** when it takes its arguments one at a time, each time returning a new function.

## Example

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$	$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
$f\ x\ y = x + y$	$f\ x = \lambda y \rightarrow x + y$

$f\ a\ b$	$(f\ a)\ b$
$= a + b$	

# Currying

A function is **curried** when it takes its arguments one at a time, each time returning a new function.

## Example

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$	$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
---	---

$f\ x\ y = x + y$	$f\ x = \backslash y \rightarrow x + y$
-------------------	---

$f\ a\ b$	$(f\ a)\ b$
$= a + b$	$= (\backslash y \rightarrow a + y)\ b$

# Currying

A function is **curried** when it takes its arguments one at a time, each time returning a new function.

## Example

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$	$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
$f\ x\ y = x + y$	$f\ x = \backslash y \rightarrow x + y$

$f\ a\ b$	$(f\ a)\ b$
$= a + b$	$= (\backslash y \rightarrow a + y)\ b$
	$= a + b$



# Currying

A function is **curried** when it takes its arguments one at a time, each time returning a new function.

## Example

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$	$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$
$f\ x\ y = x + y$	$f\ x = \backslash y \rightarrow x + y$

$f\ a\ b$	$(f\ a)\ b$
$= a + b$	$= (\backslash y \rightarrow a + y)\ b$
	$= a + b$

Any function of two arguments can be viewed as  
a function of the first argument that returns  
a function of the second argument.

# Anonymous functions (lambdas)

An **anonymous function** (or **lambda abstraction**) is a function without a name.

# Anonymous functions (lambdas)

An **anonymous function** (or **lambda abstraction**) is a function without a name.

## Examples

$(\lambda x. x + 1) 4$

# Anonymous functions (lambdas)

An **anonymous function** (or **lambda abstraction**) is a function without a name.

## Examples

$$(\lambda x \rightarrow x + 1) \ 4$$
$$= 5$$
$$(\lambda x \ y \rightarrow x + y) \ 3 \ 5$$

# Anonymous functions (lambdas)

An **anonymous function** (or **lambda abstraction**) is a function without a name.

## Examples

$$(\lambda x \rightarrow x + 1) \ 4$$
$$= 5$$
$$(\lambda x \ y \rightarrow x + y) \ 3 \ 5$$
$$= 8$$

# Anonymous functions (lambdas)

An **anonymous function** (or **lambda abstraction**) is a function without a name.

## Examples

```
(\x -> x + 1) 4  
= 5
```

```
(\x y -> x + y) 3 5  
= 8
```

What is the type of `\n -> iter n succ` where  
`iter :: Integer -> (a -> a) -> (a -> a)`  
`succ :: Integer -> Integer`

# Anonymous functions (lambdas)

An **anonymous function** (or **lambda abstraction**) is a function without a name.

## Examples

```
(\x -> x + 1) 4  
= 5
```

```
(\x y -> x + y) 3 5  
= 8
```

What is the type of `\n -> iter n succ` where  
`iter :: Integer -> (a -> a) -> (a -> a)`  
`succ :: Integer -> Integer`

`Integer -> (Integer -> Integer)`

## Partial application

Every function of  $n$  parameters can be applied to less than  $n$  arguments.



## Partial application

Every function of  $n$  parameters can be applied to less than  $n$  arguments.

A function is **partially applied** when some arguments have already been applied to a function (some parameters are already *fixed*), but some parameters are missing.

# Partial application

Every function of  $n$  parameters can be applied to less than  $n$  arguments.

A function is **partially applied** when some arguments have already been applied to a function (some parameters are already *fixed*), but some parameters are missing.

Partially applied?

- elem 5

# Partial application

Every function of  $n$  parameters can be applied to less than  $n$  arguments.

A function is **partially applied** when some arguments have already been applied to a function (some parameters are already *fixed*), but some parameters are missing.

## Partially applied?

- `elem 5` yes
- `('elem' [1..5]) 0`

# Partial application

Every function of  $n$  parameters can be applied to less than  $n$  arguments.

A function is **partially applied** when some arguments have already been applied to a function (some parameters are already *fixed*), but some parameters are missing.

## Partially applied?

- `elem 5` yes
- `('elem' [1..5]) 0` no

# Partial application

Every function of  $n$  parameters can be applied to less than  $n$  arguments.

A function is **partially applied** when some arguments have already been applied to a function (some parameters are already *fixed*), but some parameters are missing.

## Partially applied?

- `elem 5` yes
- `('elem' [1..5]) 0` no

Expressions of the form `(infixop expr)` or `(expr infixop)` are called **sections**.

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
- $\text{const} :: a \rightarrow (b \rightarrow a)$



# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
- $\text{const} :: a \rightarrow (b \rightarrow a)$
- $\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$
- $\text{const} :: a \rightarrow (b \rightarrow a)$
- $\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$
- $\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c)$

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `const :: a -> (b -> a)`
- `curry :: ((a,b) -> c) -> (a -> b -> c)`
- `uncurry :: (a -> b -> c) -> ((a,b) -> c)`
- `filter :: (a -> Bool) -> [a] -> [a]`

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `const :: a -> (b -> a)`
- `curry :: ((a,b) -> c) -> (a -> b -> c)`
- `uncurry :: (a -> b -> c) -> ((a,b) -> c)`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `map :: (a -> b) -> [a] -> [b]`

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `const :: a -> (b -> a)`
- `curry :: ((a,b) -> c) -> (a -> b -> c)`
- `uncurry :: (a -> b -> c) -> ((a,b) -> c)`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `map :: (a -> b) -> [a] -> [b]`
- `all, any :: (a -> Bool) -> [a] -> Bool`

# Higher-order functions

A **higher-order function** is a function that takes another function as an argument or returns a function.

## Examples

- `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
- `const :: a -> (b -> a)`
- `curry :: ((a,b) -> c) -> (a -> b -> c)`
- `uncurry :: (a -> b -> c) -> ((a,b) -> c)`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `map :: (a -> b) -> [a] -> [b]`
- `all, any :: (a -> Bool) -> [a] -> Bool`
- `takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]`

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
```

```
foldr f a [] = a
```

```
foldr f a (x:xs) = f x (foldr f a xs)
```

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
```

```
foldr f a []      = a
```

```
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?



# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
```

```
foldr f a []      = a
```

```
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?

```
foldr (+) 0 [1,2,3]
```

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a []      = a
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?

```
foldr (+) 0 [1,2,3]
= 1 + foldr (+) 0 [2,3]
```

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a []      = a
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?

```
foldr (+) 0 [1,2,3]
= 1 + foldr (+) 0 [2,3]
= 1 + (2 + foldr (+) 0 [3])
```

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
```

```
foldr f a [] = a
```

```
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?

```
foldr (+) 0 [1,2,3]
```

```
= 1 + foldr (+) 0 [2,3]
```

```
= 1 + (2 + foldr (+) 0 [3])
```

```
= 1 + (2 + (3 + foldr (+) 0 []))
```

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
```

```
foldr f a [] = a
```

```
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?

```
foldr (+) 0 [1,2,3]
```

```
= 1 + foldr (+) 0 [2,3]
```

```
= 1 + (2 + foldr (+) 0 [3])
```

```
= 1 + (2 + (3 + foldr (+) 0 []))
```

```
= 1 + (2 + (3 + 0))
```

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a []      = a
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?

```
foldr (+) 0 [1,2,3]
= 1 + foldr (+) 0 [2,3]
= 1 + (2 + foldr (+) 0 [3])
= 1 + (2 + (3 + foldr (+) 0 []))
= 1 + (2 + (3 + 0))
= 1 + (2 + 3)
```

# Fold

Folding is the most elementary way  
of combining elements of a list.

Right-associative (foldr):

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a []      = a
foldr f a (x:xs) = f x (foldr f a xs)
```

Why is this right-associative?

```
foldr (+) 0 [1,2,3]
= 1 + foldr (+) 0 [2,3]
= 1 + (2 + foldr (+) 0 [3])
= 1 + (2 + (3 + foldr (+) 0 []))
= 1 + (2 + (3 + 0))
= 1 + (2 + 3)
= 1 + 5 = 6
```

# Plan

## Types

- Type aliases

- Type Classes

- Algebraic Data Types

- Modules, Abstract Data Types

- Type inference



## Type aliases

Allows the renaming of a more complex type expression.

# Type aliases

Allows the renaming of a more complex type expression.

## Examples

```
type String = [Char]  
type List a = [a]
```

# Type Classes

Type classes are collections of types that implement some fixed set of functions.

# Type Classes

Type classes are collections of types that implement some fixed set of functions.

Similar concepts are commonly called *interfaces*.

# Type Classes

Type classes are collections of types that implement some fixed set of functions.

Similar concepts are commonly called *interfaces*.

Creating and using a type class:

# Type Classes

Type classes are collections of types that implement some fixed set of functions.

Similar concepts are commonly called *interfaces*.

Creating and using a type class:

1. creating a type class  $\sim$  creating an interface (define set of functions)

# Type Classes

Type classes are collections of types that implement some fixed set of functions.

Similar concepts are commonly called *interfaces*.

Creating and using a type class:

1. creating a type class  $\sim$  creating an interface (define set of functions)
2. instantiating a type class  $\sim$  implementing an interface (implement a set of functions for a member of a type class)

# Type Classes

## Examples

```
class Eq a where
```



# Type Classes

## Examples

```
class Eq a where  
    (==) :: a -> a -> Bool
```

# Type Classes

## Examples

```
class Eq a where  
    (==) :: a -> a -> Bool
```

```
instance Eq Bool where
```

# Type Classes

## Examples

```
class Eq a where  
    (==) :: a -> a -> Bool
```

```
instance Eq Bool where  
    True  == True  = True
```

# Type Classes

## Examples

```
class Eq a where  
    (==) :: a -> a -> Bool
```

```
instance Eq Bool where  
    True  == True  = True  
    False == False = True
```

# Type Classes

## Examples

```
class Eq a where  
    (==) :: a -> a -> Bool
```

```
instance Eq Bool where  
    True  == True   = True  
    False == False  = True  
    _     == _      = False
```

## Constrained instances

Instances of type classes can be constrained.

## Constrained instances

Instances of type classes can be constrained.

### Example

```
instance (Eq a) => Eq [a] where
```

# Constrained instances

Instances of type classes can be constrained.

## Example

```
instance (Eq a) => Eq [a] where  
    [] == [] = True
```



## Constrained instances

Instances of type classes can be constrained.

### Example

```
instance (Eq a) => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
```

## Constrained instances

Instances of type classes can be constrained.

### Example

```
instance (Eq a) => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _ == _ = False
```

# Subclasses

## Example

```
class (Eq a) => Ord a where  
  (<=), (<), (>=), (>) :: a -> a -> Bool
```

Class `Ord` inherits all functions of class `Eq`.

# Subclasses

## Example

```
class (Eq a) => Ord a where  
  (<=), (<), (>=), (>) :: a -> a -> Bool
```

Class `Ord` inherits all functions of class `Eq`.

Before instantiating a subclass with a type, the type must be an instance of all *"superclasses"*.

# Subclasses

## Example

```
class (Eq a) => Ord a where  
  (<=), (<), (>=), (>) :: a -> a -> Bool
```

Class Ord inherits all functions of class Eq.

Before instantiating a subclass with a type, the type must be an instance of all *"superclasses"*.

```
instance Ord Bool where  
  b1 <= b2 = not b1 || b2  
  b1 < b2 = b1 <= b2 && not(b1 == b2)
```

# Algebraic Data Types

A custom datatype with one or more constructors.

# Algebraic Data Types

A custom datatype with one or more constructors.

`data type  $a_1 \dots a_n = \text{constructor } a_k \dots a_l \mid \dots$`

# Algebraic Data Types

A custom datatype with one or more constructors.

`data type  $a_1 \dots a_n = \text{constructor } a_k \dots a_l \mid \dots$`

Constructors are

- a *prefix operator* starting with a capital letter; or
- an *infix operator* starting with `:`.



# Algebraic Data Types

## Examples

```
data Bool = False | True
```

# Algebraic Data Types

## Examples

```
data Bool = False | True
```

```
data Maybe a = Nothing | Just a  
  deriving (Eq, Show)
```

# Algebraic Data Types

## Examples

```
data Bool = False | True
```

```
data Maybe a = Nothing | Just a  
  deriving (Eq, Show)
```

```
data Nat = Zero | Suc Nat  
  deriving (Eq, Show)
```

# Algebraic Data Types

## Examples

```
data Bool = False | True
```

```
data Maybe a = Nothing | Just a  
  deriving (Eq, Show)
```

```
data Nat = Zero | Suc Nat  
  deriving (Eq, Show)
```

```
data [a] = [] | (:) a [a]  
  deriving Eq
```

# Algebraic Data Types

## Examples

```
data Bool = False | True
```

```
data Maybe a = Nothing | Just a  
  deriving (Eq, Show)
```

```
data Nat = Zero | Suc Nat  
  deriving (Eq, Show)
```

```
data [a] = [] | (:) a [a]  
  deriving Eq
```

```
data Tree a = Empty | Node a (Tree a) (Tree a)  
  deriving (Eq, Show)
```

# Algebraic Data Types

Terminology:

# Algebraic Data Types

Terminology:

- a  *$n$ -ary constructor* is a function that unambiguously constructs values of a type encapsulating  $n$  arguments.

# Algebraic Data Types

Terminology:

- a  *$n$ -ary constructor* is a function that unambiguously constructs values of a type encapsulating  $n$  arguments.
- nullary constructors are also called *constants*.



# Algebraic Data Types

## Terminology:

- a  *$n$ -ary constructor* is a function that unambiguously constructs values of a type encapsulating  $n$  arguments.
- nullary constructors are also called *constants*.
- a type that expects a *type argument* is called a *parametrized type*.

# Algebraic Data Types

## Terminology:

- a  ***$n$ -ary constructor*** is a function that unambiguously constructs values of a type encapsulating  $n$  arguments.
- nullary constructors are also called ***constants***.
- a type that expects a *type argument* is called a *parametrized type*.
- ***data constructors*** are used at the *term level*, ***type constructors*** are used at the *type level*.

# Algebraic Data Types

A datatype can be thought of as the set of possible values of that type.

# Algebraic Data Types

A datatype can be thought of as the set of possible values of that type.

- the **cardinality** of a datatype is the number of all its possible values.

# Algebraic Data Types

A datatype can be thought of as the set of possible values of that type.

- the **cardinality** of a datatype is the number of all its possible values.
- a **sum type** is a type with more than one constructor (similar to a logical  $\vee$ ).

# Algebraic Data Types

A datatype can be thought of as the set of possible values of that type.

- the **cardinality** of a datatype is the number of all its possible values.
- a **sum type** is a type with more than one constructor (similar to a logical  $\vee$ ).
- a **product type** is a type whose data constructor takes more than one argument (similar to a logical  $\wedge$ ).

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
```



## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
find _ Empty = False
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
find _ Empty = False
find x (Node a l r)
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
find _ Empty = False
find x (Node a l r)
  | x < a      = find x l
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
find _ Empty = False
find x (Node a l r)
  | x < a      = find x l
  | a < x      = find x r
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
```

```
find _ Empty = False
```

```
find x (Node a l r)
```

```
  | x < a      = find x l
```

```
  | a < x      = find x r
```

```
  | otherwise = True
```

```
insert :: Ord a => a -> Tree a -> Tree a
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
```

```
find _ Empty = False
```

```
find x (Node a l r)
```

```
  | x < a      = find x l
```

```
  | a < x      = find x r
```

```
  | otherwise = True
```

```
insert :: Ord a => a -> Tree a -> Tree a
```

```
insert x Empty = Node x Empty Empty
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
```

```
find _ Empty = False
```

```
find x (Node a l r)
```

```
  | x < a      = find x l
```

```
  | a < x      = find x r
```

```
  | otherwise = True
```

```
insert :: Ord a => a -> Tree a -> Tree a
```

```
insert x Empty = Node x Empty Empty
```

```
insert x (Node a l r)
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
```

```
find _ Empty = False
```

```
find x (Node a l r)
```

```
  | x < a      = find x l
```

```
  | a < x      = find x r
```

```
  | otherwise = True
```

```
insert :: Ord a => a -> Tree a -> Tree a
```

```
insert x Empty = Node x Empty Empty
```

```
insert x (Node a l r)
```

```
  | x < a      = Node a (insert x l) r
```



## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
```

```
find _ Empty = False
```

```
find x (Node a l r)
```

```
  | x < a      = find x l
```

```
  | a < x      = find x r
```

```
  | otherwise = True
```

```
insert :: Ord a => a -> Tree a -> Tree a
```

```
insert x Empty = Node x Empty Empty
```

```
insert x (Node a l r)
```

```
  | x < a      = Node a (insert x l) r
```

```
  | a < x      = Node a l (insert x r)
```

## Pattern matching

Pattern matching works just the same for custom constructors as for predefined constructors.

### Examples

```
find :: Ord a => a -> Tree a -> Bool
find _ Empty = False
find x (Node a l r)
  | x < a      = find x l
  | a < x      = find x r
  | otherwise = True
```

```
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty = Node x Empty Empty
insert x (Node a l r)
  | x < a      = Node a (insert x l) r
  | a < x      = Node a l (insert x r)
  | otherwise = Node a l r
```

# Modules

Collection of type, function, class and other definitions.

# Modules

Collection of type, function, class and other definitions.

## Examples

```
module M where  
exports everything defined in M
```

# Modules

Collection of type, function, class and other definitions.

## Examples

```
module M where  
exports everything defined in M
```

```
module M (T, f, ...) where  
exports only T, f, ...
```

## Exporting data types

```
module M (T) where
```

```
data T = ...
```

```
exports only T but not its constructors
```

## Exporting data types

```
module M (T) where
```

```
data T = ...
```

```
exports only T but not its constructors
```

```
module M (T(C,D,...)) where
```

```
data T = ...
```

```
exports T and its constructors C, D, ...
```

## Exporting data types

```
module M (T) where
```

```
data T = ...
```

exports only T but not its constructors

```
module M (T(C,D,...)) where
```

```
data T = ...
```

exports T and its constructors C, D, ...

```
module M (T(..)) where
```

```
data T = ...
```

exports T and all its constructors



## Exporting data types

```
module M (T) where  
data T = ...  
exports only T but not its constructors
```

```
module M (T(C,D,...)) where  
data T = ...  
exports T and its constructors C, D, ...
```

```
module M (T(..)) where  
data T = ...  
exports T and all its constructors
```

Not allowed (why?):

```
module M (T,C,D) where
```

## Exporting data types

```
module M (T) where
data T = ...
exports only T but not its constructors
```

```
module M (T(C,D,...)) where
data T = ...
exports T and its constructors C, D, ...
```

```
module M (T(..)) where
data T = ...
exports T and all its constructors
```

Not allowed (why?):

```
module M (T,C,D) where
```

Constructors could have the same name as a type.

# Abstract Data Types

Hides data representation by wrapping data in a constructor that is not exported.

# Abstract Data Types

## Example

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty :: Set a
insert :: Eq a => a -> Set a -> Set a
isin :: Eq a => a -> Set a -> Bool
size :: Set a -> Int
```

# Abstract Data Types

## Example

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty :: Set a
insert :: Eq a => a -> Set a -> Set a
isin :: Eq a => a -> Set a -> Bool
size :: Set a -> Int
-- Implementation
newtype Set a = Set [a]
```

# Abstract Data Types

## Example

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty :: Set a
insert :: Eq a => a -> Set a -> Set a
isin :: Eq a => a -> Set a -> Bool
size :: Set a -> Int
-- Implementation
newtype Set a = Set [a]

empty = Set []
```

# Abstract Data Types

## Example

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty :: Set a
insert :: Eq a => a -> Set a -> Set a
isin :: Eq a => a -> Set a -> Bool
size :: Set a -> Int
-- Implementation
newtype Set a = Set [a]

empty = Set []
insert x (Set xs) = Set (if elem x xs then xs else x:xs)
```

# Abstract Data Types

## Example

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty :: Set a
insert :: Eq a => a -> Set a -> Set a
isin :: Eq a => a -> Set a -> Bool
size :: Set a -> Int
-- Implementation
newtype Set a = Set [a]

empty = Set []
insert x (Set xs) = Set (if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
```



# Abstract Data Types

## Example

```
module Set (Set, empty, insert, isin, size) where
-- Interface
empty :: Set a
insert :: Eq a => a -> Set a -> Set a
isin :: Eq a => a -> Set a -> Bool
size :: Set a -> Int
-- Implementation
newtype Set a = Set [a]

empty = Set []
insert x (Set xs) = Set (if elem x xs then xs else x:xs)
isin x (Set xs) = elem x xs
size (Set xs) = length xs
```

## type vs data vs newtype

- type is used to create type aliases

## type vs data vs newtype

- type is used to create type aliases
- data is used to create algebraic data types (types with a custom shape)

## type vs data vs newtype

- `type` is used to create type aliases
- `data` is used to create algebraic data types (types with a custom shape)
- `newtype` is used to create a custom constructor for a single type without adding any runtime overhead

# Type inference

Inferring/reconstructing the type of an expression.

# Type inference

Inferring/reconstructing the type of an expression.

Given an expression  $e$ .

# Type inference

Inferring/reconstructing the type of an expression.

Given an expression  $e$ .

1. give all variables in  $e$  distinct type variables

# Type inference

Inferring/reconstructing the type of an expression.

Given an expression  $e$ .

1. give all variables in  $e$  distinct type variables
2. give each function  $f :: T$  in  $e$  a new general type with fresh type variables



# Type inference

Inferring/reconstructing the type of an expression.

Given an expression  $e$ .

1. give all variables in  $e$  distinct type variables
2. give each function  $f :: T$  in  $e$  a new general type with fresh type variables
3. for each sub-expression in  $e$  set up an equation linking the type of parameters and arguments

# Type inference

Inferring/reconstructing the type of an expression.

Given an expression  $e$ .

1. give all variables in  $e$  distinct type variables
2. give each function  $f :: T$  in  $e$  a new general type with fresh type variables
3. for each sub-expression in  $e$  set up an equation linking the type of parameters and arguments
4. simplify the set of equations by replacing equivalences

# Type inference

## Example

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

# Type inference

## Example

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 1

# Type inference

## Example

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 1

1.  $u :: a$

2.  $v :: b$

Step 2

# Type inference

## Example

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 1

1.  $u :: a$

2.  $v :: b$

Step 2

1.  $\text{head} :: [c] \rightarrow c$

# Type inference

## Example

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 1

1.  $u :: a$
2.  $v :: b$

Step 2

1.  $\text{head} :: [c] \rightarrow c$
2.  $\text{concat} :: [[d]] \rightarrow [d]$

# Type inference

## Example

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 1

1.  $u :: a$
2.  $v :: b$

Step 2

1.  $\text{head} :: [c] \rightarrow c$
2.  $\text{concat} :: [[d]] \rightarrow [d]$
3.  $\text{last} :: [e] \rightarrow e$



# Type inference

## Example

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 1

1.  $u :: a$
2.  $v :: b$

Step 2

1.  $\text{head} :: [c] \rightarrow c$
2.  $\text{concat} :: [[d]] \rightarrow [d]$
3.  $\text{last} :: [e] \rightarrow e$
4.  $\text{min} :: \text{Ord}\ f \Rightarrow f \rightarrow f \rightarrow f$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 3

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 3

1. from  $\text{head}\ u$  derive  $[c] = a$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 3

1. from  $\text{head}\ u$  derive  $[c] = a$
2. from  $\text{concat}\ v$  derive  $[[d]] = b$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 3

1. from  $\text{head}\ u$  derive  $[c] = a$
2. from  $\text{concat}\ v$  derive  $[[d]] = b$
3. from  $\text{last}\ (\text{concat}\ v)$  derive  $[e] = [d]$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Step 3

1. from  $\text{head}\ u$  derive  $[c] = a$
2. from  $\text{concat}\ v$  derive  $[[d]] = b$
3. from  $\text{last}\ (\text{concat}\ v)$  derive  $[e] = [d]$
4. from  $\text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$  derive  $f = c$  and  $f = e$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update



# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update

- $u :: [c]$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update

# Type inference

## Example (cont.)

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update
  - $v :: [[d]]$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update
  - $v :: [[d]]$
3. apply  $[e] = [d]$  to get  $e = d$  and update

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min } (\text{head } u)\ (\text{last } (\text{concat } v))$

Goal  $f :: \text{Ord } f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update
  - $v :: [[d]]$
3. apply  $[e] = [d]$  to get  $e = d$  and update
  - $v :: [[e]]$

# Type inference

## Example (cont.)

Given  $f \ u \ v = \text{min} \ (\text{head } u) \ (\text{last} \ (\text{concat } v))$

Goal  $f :: \text{Ord } f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update
  - $v :: [[d]]$
3. apply  $[e] = [d]$  to get  $e = d$  and update
  - $v :: [[e]]$
  - $\text{concat} :: [[e]] \rightarrow [e]$

# Type inference

## Example (cont.)

Given  $f \ u \ v = \text{min } (\text{head } u) \ (\text{last } (\text{concat } v))$

Goal  $f :: \text{Ord } f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update
  - $v :: [[d]]$
3. apply  $[e] = [d]$  to get  $e = d$  and update
  - $v :: [[e]]$
  - $\text{concat} :: [[e]] \rightarrow [e]$
4. apply  $f = c$  and update

# Type inference

## Example (cont.)

Given  $f \ u \ v = \text{min} \ (\text{head } u) \ (\text{last} \ (\text{concat } v))$

Goal  $f :: \text{Ord } f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update
  - $v :: [[d]]$
3. apply  $[e] = [d]$  to get  $e = d$  and update
  - $v :: [[e]]$
  - $\text{concat} :: [[e]] \rightarrow [e]$
4. apply  $f = c$  and update
  - $u :: [f]$



# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min } (\text{head } u)\ (\text{last } (\text{concat } v))$

Goal  $f :: \text{Ord } f \Rightarrow a \rightarrow b \rightarrow f$

Step 4

1. apply  $[c] = a$  and update
  - $u :: [c]$
2. apply  $[[d]] = b$  and update
  - $v :: [[d]]$
3. apply  $[e] = [d]$  to get  $e = d$  and update
  - $v :: [[e]]$
  - $\text{concat} :: [[e]] \rightarrow [e]$
4. apply  $f = c$  and update
  - $u :: [f]$
  - $\text{head} :: [f] \rightarrow f$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4 (cont.)

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min } (\text{head } u) (\text{last } (\text{concat } v))$

Goal  $f :: \text{Ord } f \Rightarrow a \rightarrow b \rightarrow f$

Step 4 (cont.)

1. apply  $f = e$  and update

# Type inference

## Example (cont.)

Given  $f\ u\ v = \text{min}\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4 (cont.)

1. apply  $f = e$  and update
  - $v :: [[f]]$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4 (cont.)

1. apply  $f = e$  and update

- $v :: [[f]]$
- $\text{concat} :: [[f]] \rightarrow [f]$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4 (cont.)

1. apply  $f = e$  and update

- $v :: [[f]]$
- $\text{concat} :: [[f]] \rightarrow [f]$
- $\text{last} :: [[f]] \rightarrow [f]$

# Type inference

## Example (cont.)

Given  $f\ u\ v = \min\ (\text{head}\ u)\ (\text{last}\ (\text{concat}\ v))$

Goal  $f :: \text{Ord}\ f \Rightarrow a \rightarrow b \rightarrow f$

Step 4 (cont.)

1. apply  $f = e$  and update

- $v :: [[f]]$
- $\text{concat} :: [[f]] \rightarrow [f]$
- $\text{last} :: [[f]] \rightarrow [f]$

2. no further simplification possible,

return  $f :: \text{Ord}\ f \Rightarrow [f] \rightarrow [[f]] \rightarrow f$

# Plan

## Proofs

- Structural induction

- Case analysis

- Generalization

- Extensionality

- Computation induction



# Structural induction

Induction on the structural definition of a datatype

# Structural induction

## Induction on the structural definition of a datatype

To prove property  $P(x)$  for all finite values  $x$  of type  $T$ ,  
prove  $P(C)$  for each constructor  $C$  of  $T$ .

# Structural induction

## Induction on the structural definition of a datatype

To prove property  $P(x)$  for all finite values  $x$  of type  $T$ ,  
prove  $P(C)$  for each constructor  $C$  of  $T$ .

- **base cases** are represented by proofs for non-recursive constructors

# Structural induction

## Induction on the structural definition of a datatype

To prove property  $P(x)$  for all finite values  $x$  of type  $T$ ,  
prove  $P(C)$  for each constructor  $C$  of  $T$ .

- **base cases** are represented by proofs for non-recursive constructors
- **inductive cases** are represented by proofs for recursive constructors

# Structural induction

## Induction on the structural definition of a datatype

To prove property  $P(x)$  for all finite values  $x$  of type  $T$ ,  
prove  $P(C)$  for each constructor  $C$  of  $T$ .

- **base cases** are represented by proofs for non-recursive constructors
- **inductive cases** are represented by proofs for recursive constructors

Each recursive type parameter has a separate induction hypothesis.  
(Why?)

# Structural induction on trees

## Example

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

# Structural induction on trees

## Example

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

```
mirror Leaf = Leaf
```

```
mirror (Node l v r) = Node (mirror r) v (mirror l)
```

```
id x = x
```

```
(f . g) x = f (g x)
```

# Structural induction on trees

## Example

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

```
mirror Leaf = Leaf
```

```
mirror (Node l v r) = Node (mirror r) v (mirror l)
```

```
id x = x
```

```
(f . g) x = f (g x)
```

```
Prove (mirror . mirror) t .=. id t.
```



# Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = . \ \text{id} \ t$

# Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = . \ \text{id} \ t$

Proof by induction on Tree t

# Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = \ \text{id} \ t$

Proof by induction on Tree t

Case Leaf

# Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = . \ \text{id} \ t$

Proof by induction on Tree t

Case Leaf

To show:  $(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf} \ . = . \ \text{id} \ \text{Leaf}$

Proof

$(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf}$

## Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = . \ \text{id} \ t$

Proof by induction on Tree  $t$

## Case Leaf

To show:  $(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf} \ . = . \ \text{id} \ \text{Leaf}$

Proof

```

      (mirror . mirror) Leaf
(by def .)    .= mirror (mirror Leaf)

```

# Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = . \ \text{id} \ t$

Proof by induction on Tree  $t$

Case Leaf

To show:  $(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf} \ . = . \ \text{id} \ \text{Leaf}$

Proof

	$(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf}$
$(\text{by def } .)$	$. = . \ \underline{\text{mirror} (\text{mirror Leaf})}$
$(\text{by def mirror})$	$. = . \ \underline{\text{mirror Leaf}}$

# Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = . \ \text{id} \ t$

Proof by induction on Tree  $t$

### Case Leaf

To show:  $(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf} \ . = . \ \text{id} \ \text{Leaf}$

Proof

	$(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf}$
$(\text{by def } .)$	$. = . \ \underline{\text{mirror} (\text{mirror Leaf})}$
$(\text{by def mirror})$	$. = . \ \underline{\text{mirror Leaf}}$
$(\text{by def mirror})$	$. = . \ \underline{\text{Leaf}}$

# Structural induction on trees

## Example (cont.)

Lemma:  $(\text{mirror} \ . \ \text{mirror}) \ t \ . = . \ \text{id} \ t$

Proof by induction on Tree t

### Case Leaf

To show:  $(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf} \ . = . \ \text{id} \ \text{Leaf}$

Proof

	$(\text{mirror} \ . \ \text{mirror}) \ \text{Leaf}$
$(\text{by def } .)$	$. = . \ \underline{\text{mirror} (\text{mirror Leaf})}$
$(\text{by def mirror})$	$. = . \ \underline{\text{mirror Leaf}}$
$(\text{by def mirror})$	$. = . \ \underline{\text{Leaf}}$
$(\text{by def id})$	$. = . \ \underline{\text{id Leaf}}$

QED



# Structural induction on trees

## Example (cont.)

Case Node l v r

# Structural induction on trees

## Example (cont.)

### Case Node l v r

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $\text{.=. id } (\text{Node } l \ v \ r)$

# Structural induction on trees

## Example (cont.)

### Case Node l v r

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $\text{.=. id } (\text{Node } l \ v \ r)$

IH1:        $(\text{mirror} \ . \ \text{mirror}) \ l \ \text{.=. id } l$

IH2:        $(\text{mirror} \ . \ \text{mirror}) \ r \ \text{.=. id } r$

Proof

$(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$

# Structural induction on trees

## Example (cont.)

### Case Node l v r

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $\text{.=. id } (\text{Node } l \ v \ r)$

IH1:  $(\text{mirror} \ . \ \text{mirror}) \ l \text{ .=. id } l$

IH2:  $(\text{mirror} \ . \ \text{mirror}) \ r \text{ .=. id } r$

Proof

$(\text{by def } \ .) \quad (\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
 $\text{.=. } \underline{\text{mirror } (\text{mirror } (\text{Node } l \ v \ r))}$

# Structural induction on trees

## Example (cont.)

### Case Node l v r

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $.\ = \ \text{id} \ (\text{Node } l \ v \ r)$

IH1:        $(\text{mirror} \ . \ \text{mirror}) \ l \ .\ = \ \text{id} \ l$

IH2:        $(\text{mirror} \ . \ \text{mirror}) \ r \ .\ = \ \text{id} \ r$

Proof

$(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
(by def .)                    $.\ = \ \underline{\text{mirror} \ (\text{mirror} \ (\text{Node } l \ v \ r))}$   
(by def mirror)  
 $.\ = \ \text{mirror} \ (\underline{\text{Node} \ (\text{mirror} \ r) \ v \ (\text{mirror} \ l)})$

# Structural induction on trees

## Example (cont.)

### Case Node l v r

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $.\ = \ \text{id} \ (\text{Node } l \ v \ r)$

IH1:        $(\text{mirror} \ . \ \text{mirror}) \ l \ .\ = \ \text{id} \ l$

IH2:        $(\text{mirror} \ . \ \text{mirror}) \ r \ .\ = \ \text{id} \ r$

Proof

$(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
(by def .)                    $.\ = \ \underline{\text{mirror} \ (\text{mirror} \ (\text{Node } l \ v \ r))}$   
(by def mirror)  
 $.\ = \ \underline{\text{mirror} \ (\text{Node} \ (\text{mirror} \ r) \ v \ (\text{mirror} \ l))}$   
(by def mirror)  
 $.\ = \ \underline{\text{Node} \ (\text{mirror} \ (\text{mirror} \ l)) \ v \ (\text{mirror} \ (\text{mirror} \ r))}$

# Structural induction on trees

## Example (cont.)

### Case Node l v r

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $.\ = \ \text{id} \ (\text{Node } l \ v \ r)$

IH1:        $(\text{mirror} \ . \ \text{mirror}) \ l \ .\ = \ \text{id} \ l$

IH2:        $(\text{mirror} \ . \ \text{mirror}) \ r \ .\ = \ \text{id} \ r$

Proof

$(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
(by def .)                    $.\ = \ \underline{\text{mirror} \ (\text{mirror} \ (\text{Node } l \ v \ r))}$   
(by def mirror)  
 $.\ = \ \underline{\text{mirror} \ (\text{Node} \ (\text{mirror} \ r) \ v \ (\text{mirror} \ l))}$   
(by def mirror)  
 $.\ = \ \underline{\text{Node} \ (\text{mirror} \ (\text{mirror} \ l)) \ v \ (\text{mirror} \ (\text{mirror} \ r))}$   
(by def .)  
 $.\ = \ \text{Node} \ ((\underline{\text{mirror} \ . \ \text{mirror}}) \ l) \ v \ (\text{mirror} \ (\text{mirror} \ r))$

# Structural induction on trees

## Example (cont.)

### Case Node l v r

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $\text{.=. id } (\text{Node } l \ v \ r)$

IH1:        $(\text{mirror} \ . \ \text{mirror}) \ l \ \text{.=. id } l$

IH2:        $(\text{mirror} \ . \ \text{mirror}) \ r \ \text{.=. id } r$

Proof

```

                                (mirror . mirror) (Node l v r)
(by def .)                    .=. mirror (mirror (Node l v r))
(by def mirror)
.=. mirror (Node (mirror r) v (mirror l))
(by def mirror)
.=. Node (mirror (mirror l)) v (mirror (mirror r))
(by def .)
.=. Node ((mirror . mirror) l) v (mirror (mirror r))
(by def .)
.=. Node ((mirror . mirror) l) v ((mirror . mirror) r)
```



# Structural induction on trees

## Example (cont.)

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$   
           $.\! = \text{id} \ (\text{Node } l \ v \ r)$

IH1:  $(\text{mirror} \ . \ \text{mirror}) \ l \ .\! = \text{id } l$

IH2:  $(\text{mirror} \ . \ \text{mirror}) \ r \ .\! = \text{id } r$

Proof

⋮

(by def .)

$.\! = \text{Node } ((\text{mirror} \ . \ \text{mirror}) \ l) \ v \ ((\text{mirror} \ . \ \text{mirror}) \ r)$

# Structural induction on trees

## Example (cont.)

To show: `(mirror . mirror) (Node l v r)`

`.=. id (Node l v r)`

IH1: `(mirror . mirror) l .==. id l`

IH2: `(mirror . mirror) r .==. id r`

Proof

`:`

`(by def .)`

`.=. Node ((mirror . mirror) l) v ((mirror . mirror) r)`

`(by IH1) .==. Node (id l) v ((mirror . mirror) r)`

`(by IH2) .==. Node (id l) v (id r)`

## Structural induction on trees

## Example (cont.)

To show: (mirror . mirror) (Node l v r)

```

.=. id (Node l v r)

```

IH1:  $(\text{mirror} \ . \ \text{mirror}) \ 1 \ . = . \ \text{id} \ 1$

```
IH2: (mirror . mirror) r .=. id r
```

### Proof

•

(by def .)

```

.=. Node ((mirror . mirror) l) v ((mirror . mirror) r)

```

```
(by IH1)      . = . Node (id l) v ((mirror . mirror) r)
```

$$(\text{by IH2}) \quad . = . \text{ Node } (\text{id } l) \ v \ (\text{id } r)$$

```
(by def id) . = . Node l v (id r)
```

# Structural induction on trees

## Example (cont.)

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$

$\quad \quad \quad \text{.} = \text{id} \ (\text{Node } l \ v \ r)$

IH1:  $(\text{mirror} \ . \ \text{mirror}) \ l \text{ .} = \text{id } l$

IH2:  $(\text{mirror} \ . \ \text{mirror}) \ r \text{ .} = \text{id } r$

Proof

$\vdots$

(by def .)

$\text{.} = \text{Node } ((\text{mirror} \ . \ \text{mirror}) \ l) \ v \ ((\text{mirror} \ . \ \text{mirror}) \ r)$

(by IH1)  $\text{.} = \text{Node } \underline{(\text{id } l)} \ v \ ((\text{mirror} \ . \ \text{mirror}) \ r)$

(by IH2)  $\text{.} = \text{Node } (\text{id } l) \ v \ \underline{(\text{id } r)}$

(by def id)  $\text{.} = \text{Node } \underline{l} \ v \ (\text{id } r)$

(by def id)  $\text{.} = \text{Node } l \ v \ \underline{r}$

# Structural induction on trees

## Example (cont.)

To show:  $(\text{mirror} \ . \ \text{mirror}) \ (\text{Node } l \ v \ r)$

$\quad \quad \quad \text{.} = \text{. id } (\text{Node } l \ v \ r)$

IH1:  $(\text{mirror} \ . \ \text{mirror}) \ l \text{ .} = \text{. id } l$

IH2:  $(\text{mirror} \ . \ \text{mirror}) \ r \text{ .} = \text{. id } r$

Proof

$\vdots$

(by def .)

$\text{.} = \text{. Node } ((\text{mirror} \ . \ \text{mirror}) \ l) \ v \ ((\text{mirror} \ . \ \text{mirror}) \ r)$

(by IH1)  $\text{.} = \text{. Node } (\underline{\text{id } l}) \ v \ ((\text{mirror} \ . \ \text{mirror}) \ r)$

(by IH2)  $\text{.} = \text{. Node } (\text{id } l) \ v \ (\underline{\text{id } r})$

(by def id)  $\text{.} = \text{. Node } \underline{l} \ v \ (\text{id } r)$

(by def id)  $\text{.} = \text{. Node } l \ v \ \underline{r}$

(by def id)  $\text{.} = \text{. } \underline{\text{id } (\text{Node } l \ v \ r)}$

QED

QED

# Structural induction on lists

Definition of a list:

# Structural induction on lists

Definition of a list:

```
data [a] = [] | a : [a]
```

# Structural induction on lists

Definition of a list:

`data [a] = [] | a : [a]`

To prove property  $P(xs)$  for all finite lists  $xs$

- Base case: Prove  $P([])$
- Inductive case: Prove  $P(xs) \implies P(x:xs)$



# Structural induction on lists

Definition of a list:

`data [a] = [] | a : [a]`

To prove property  $P(xs)$  for all finite lists  $xs$

- Base case: Prove  $P([])$
- Inductive case: Prove  $P(xs) \implies P(x:xs)$

Structural induction on lists  
are inductions on the length of a list

## Case analysis

For conditionals consider separate proofs for the cases True and False.

## Case analysis

For conditionals consider separate proofs for the cases True and False.

### Example

To show:  $\text{if } x < y \text{ then } A \text{ else } B \text{ } \dot{=} \text{ } f \ x \ y$

## Case analysis

For conditionals consider separate proofs for the cases True and False.

### Example

To show:  $\text{if } x < y \text{ then } A \text{ else } B \text{ } \dot{=} \text{ } f \ x \ y$

Proof by case analysis on  $\text{Bool } x < y$

Case True

Assumption:  $x < y \text{ } \dot{=} \text{ } \text{True}$

Proof

$\text{if } x < y \text{ then } A \text{ else } B$

## Case analysis

For conditionals consider separate proofs for the cases True and False.

### Example

To show:  $\text{if } x < y \text{ then } A \text{ else } B \text{ } \dot{=} \text{ } f \ x \ y$

Proof by case analysis on  $\text{Bool } x < y$

Case True

Assumption:  $x < y \text{ } \dot{=} \text{ } \text{True}$

Proof

$\text{if } x < y \text{ then } A \text{ else } B$   
(by Assumption)  $\dot{=} \text{ if True then } A \text{ else } B$

## Case analysis

For conditionals consider separate proofs for the cases True and False.

### Example

To show:  $\text{if } x < y \text{ then } A \text{ else } B \text{ } .=. \text{ f } x \ y$

Proof by case analysis on  $\text{Bool } x < y$

Case True

Assumption:  $x < y \text{ } .=. \text{ True}$

Proof

$\text{if } x < y \text{ then } A \text{ else } B$   
(by Assumption)  $.=. \text{ if True then } A \text{ else } B$   
(by ifTrue)  $.=. A$

...

QED

## Case analysis

For conditionals consider separate proofs for the cases True and False.

### Example

To show:  $\text{if } x < y \text{ then } A \text{ else } B \text{ } \dot{=} \text{ } f \ x \ y$

Proof by case analysis on  $\text{Bool } x < y$

Case True

Assumption:  $x < y \text{ } \dot{=} \text{ } \text{True}$

Proof

$\text{if } x < y \text{ then } A \text{ else } B$   
(by Assumption)  $\dot{=} \text{ if True then } A \text{ else } B$   
(by ifTrue)  $\dot{=} A$

...

QED

Case False

...

QED

# Generalization

When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.



# Generalization

When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

## Example

Consider a structural induction on  $xs$   
with the IH  $f\ xs\ ys\ . = .\ g\ xs\ ys$ .

# Generalization

When using the IH, variables may be replaced by arbitrary expressions, only the induction variable must stay fixed.

## Example

Consider a structural induction on  $xs$   
with the IH  $f\ xs\ ys\ . = .\ g\ xs\ ys$ .

Then,

$$f\ xs\ ys\ . = .\ g\ xs\ ys \implies f\ xs\ []\ . = .\ g\ xs\ [].$$

# Generalization

We have to prove

# Generalization

We have to prove

- a more generalized problem than the original problem; and

# Generalization

We have to prove

- a more generalized problem than the original problem; and
- that the specific instance of our problem follows from the generalized problem.

# Extensionality

Two functions are equal  
if for all arguments they yield the same result.

# Extensionality

Two functions are equal  
if for all arguments they yield the same result.

## Example

Lemma:  $f \text{ .}. g$

Proof by extensionality with  $xs$

To show:  $f \text{ } xs \text{ .}. g \text{ } xs$

Proof by induction on List  $xs$

...

QED

QED

# Computation induction

Induction on the length of a computation



# Computation induction

## Induction on the length of a computation

To prove property  $P(x_1, \dots, x_k)$  for all  $x_1, \dots, x_k$ ,  
for every defining equation

$$f \ p_1, \dots, p_k = \dots \ f \ e_{11}, \dots, e_{1k} \ \dots \ f \ e_{n1}, \dots, e_{nk} \ \dots$$

prove  $P(e_{11}, \dots, e_{1k}), \dots, P(e_{n1}, \dots, e_{nk}) \implies P(p_1, \dots, p_k)$ .

# Computation induction

## Induction on the length of a computation

To prove property  $P(x_1, \dots, x_k)$  for all  $x_1, \dots, x_k$ ,  
for every defining equation

$$f \ p_1, \dots, p_k = \dots \ f \ e_{11}, \dots, e_{1k} \ \dots \ f \ e_{n1}, \dots, e_{nk} \ \dots$$

prove  $P(e_{11}, \dots, e_{1k}), \dots, P(e_{n1}, \dots, e_{nk}) \implies P(p_1, \dots, p_k)$ .

Also referred to as an **induction on the computation** of a function  $f$   
or **f-induction**.

# Computation induction

## Example

```
splice [] ys = ys
```

```
splice (x:xs) ys = x : splice ys xs
```

# Computation induction

## Example

`splice [] ys = ys`

`splice (x:xs) ys = x : splice ys xs`

splice-induction: To prove  $P(xs, ys)$  for all  $xs$  and  $ys$ , prove

# Computation induction

## Example

`splice [] ys = ys`

`splice (x:xs) ys = x : splice ys xs`

splice-induction: To prove  $P(xs, ys)$  for all  $xs$  and  $ys$ , prove

1.  $P([], ys)$

# Computation induction

## Example

`splice [] ys = ys`

`splice (x:xs) ys = x : splice ys xs`

splice-induction: To prove  $P(xs, ys)$  for all  $xs$  and  $ys$ , prove

1.  $P([], ys)$
2.  $P(ys, xs) \implies P(x:xs, ys)$

# Computation induction

## Example

`splice [] ys = ys`

`splice (x:xs) ys = x : splice ys xs`

splice-induction: To prove  $P(xs, ys)$  for all  $xs$  and  $ys$ , prove

1.  $P([], ys)$
2.  $P(ys, xs) \implies P(x:xs, ys)$

Prove `length (splice xs ys) == length xs + length ys`.

# Computation induction

## Example

`splice [] ys = ys`

`splice (x:xs) ys = x : splice ys xs`

splice-induction: To prove  $P(xs, ys)$  for all  $xs$  and  $ys$ , prove

1.  $P([], ys)$
2.  $P(ys, xs) \implies P(x:xs, ys)$

Prove `length (splice xs ys) == length xs + length ys`.

Structural induction does not work (why?)



# Computation induction

## Example (cont.)

Lemma: `length (splice xs ys) .=. length xs + length ys`

# Computation induction

## Example (cont.)

Lemma: `length (splice xs ys) .=. length xs + length ys`

Proof by splice-induction on `xs` and `ys`

# Computation induction

## Example (cont.)

Lemma:  $\text{length} (\text{splice } xs \text{ } ys) = \text{length } xs + \text{length } ys$

Proof by splice-induction on  $xs$  and  $ys$

Case 1

# Computation induction

## Example (cont.)

Lemma: `length (splice xs ys) == length xs + length ys`

Proof by splice-induction on `xs` and `ys`

### Case 1

To show: `length (splice [] ys) == length [] + length ys`

Proof

`length (splice [] ys)`

# Computation induction

## Example (cont.)

Lemma:  $\text{length} (\text{splice } xs \ ys) =. \text{length } xs + \text{length } ys$

Proof by splice-induction on  $xs$  and  $ys$

### Case 1

To show:  $\text{length} (\text{splice } [] \ ys) =. \text{length } [] + \text{length } ys$

Proof

$$\begin{aligned} & \text{length } (\text{splice } [] \ ys) \\ (\text{by def splice}) \quad & =. \text{length } \underline{ys} \end{aligned}$$

$$\text{length } [] + \text{length } ys$$

# Computation induction

## Example (cont.)

Lemma:  $\text{length} (\text{splice } xs \ ys) =. \text{length } xs + \text{length } ys$

Proof by splice-induction on  $xs$  and  $ys$

### Case 1

To show:  $\text{length} (\text{splice } [] \ ys) =. \text{length } [] + \text{length } ys$

Proof

$$\begin{aligned} & \text{length } (\text{splice } [] \ ys) \\ (\text{by def splice}) \quad & =. \text{length } \underline{ys} \end{aligned}$$

$$\begin{aligned} & \text{length } [] + \text{length } ys \\ (\text{by def length}) \quad & =. \underline{0} + \text{length } ys \end{aligned}$$

# Computation induction

## Example (cont.)

Lemma:  $\text{length} (\text{splice } xs \text{ } ys) =. \text{length } xs + \text{length } ys$

Proof by splice-induction on  $xs$  and  $ys$

### Case 1

To show:  $\text{length} (\text{splice } [] \text{ } ys) =. \text{length } [] + \text{length } ys$

Proof

$$\begin{aligned} & \text{length } (\text{splice } [] \text{ } ys) \\ (\text{by def splice}) & =. \text{length } \underline{ys} \end{aligned}$$

$$\begin{aligned} & \text{length } [] + \text{length } ys \\ (\text{by def length}) & =. \underline{0} + \text{length } ys \end{aligned}$$

$$(\text{by def 0}) \quad =. \underline{\text{length } ys}$$

QED

# Computation induction

Example (cont.)

Case 2



# Computation induction

## Example (cont.)

### Case 2

To show:  $\text{length} (\text{splice } (x:xs) \text{ } ys)$   
           $=. \text{length } (x:xs) + \text{length } ys$

# Computation induction

## Example (cont.)

### Case 2

To show: `length (splice (x:xs) ys)`  
          `.=.` `length (x:xs) + length ys`

IH:       `length (splice ys xs)`  
          `.=.` `length ys + length xs`

Proof

`length (splice (x:xs) ys)`

# Computation induction

## Example (cont.)

### Case 2

To show:  $\text{length} (\text{splice} (x:xs) \text{ ys})$   
           $.=.$   $\text{length} (x:xs) + \text{length} \text{ ys}$

IH:  $\text{length} (\text{splice} \text{ ys} \text{ xs})$   
      $.=.$   $\text{length} \text{ ys} + \text{length} \text{ xs}$

Proof

$\text{length} (\text{splice} (x:xs) \text{ ys})$   
(by def splice)  $.=.$   $\text{length} \text{ (x : splice ys xs)}$

# Computation induction

## Example (cont.)

### Case 2

To show:  $\text{length} (\text{splice} (x:xs) \text{ ys})$   
           $=. \text{length} (x:xs) + \text{length} \text{ ys}$

IH:  $\text{length} (\text{splice} \text{ ys} \text{ xs})$   
      $=. \text{length} \text{ ys} + \text{length} \text{ xs}$

Proof

$\text{length} (\text{splice} (x:xs) \text{ ys})$   
(by def splice)  $=. \text{length} (x : \text{splice} \text{ ys} \text{ xs})$   
(by def length)  $=. \underline{1 + \text{length} (\text{splice} \text{ ys} \text{ xs})}$

# Computation induction

## Example (cont.)

### Case 2

To show:  $\text{length} (\text{splice} (x:xs) \text{ ys})$   
           $=. \text{length} (x:xs) + \text{length} \text{ ys}$

IH:  $\text{length} (\text{splice} \text{ ys} \text{ xs})$   
      $=. \text{length} \text{ ys} + \text{length} \text{ xs}$

Proof

$$\begin{aligned} & \text{length} (\text{splice} (x:xs) \text{ ys}) \\ \text{(by def splice)} & \quad =. \text{length} (x : \text{splice} \text{ ys} \text{ xs}) \\ \text{(by def length)} & \quad =. 1 + \text{length} (\text{splice} \text{ ys} \text{ xs}) \\ \text{(by IH)} & \quad =. 1 + \underline{\text{length} \text{ ys} + \text{length} \text{ xs}} \end{aligned}$$

# Computation induction

## Example (cont.)

### Case 2

To show:  $\text{length} (\text{splice} (x:xs) \text{ ys})$   
           $=. \text{length} (x:xs) + \text{length} \text{ ys}$

IH:  $\text{length} (\text{splice} \text{ ys} \text{ xs})$   
      $=. \text{length} \text{ ys} + \text{length} \text{ xs}$

Proof

$$\begin{aligned} & \text{length} (\text{splice} (x:xs) \text{ ys}) \\ \text{(by def splice)} & \quad =. \text{length} (x : \text{splice} \text{ ys} \text{ xs}) \\ \text{(by def length)} & \quad =. 1 + \text{length} (\text{splice} \text{ ys} \text{ xs}) \\ \text{(by IH)} & \quad =. 1 + (\text{length} \text{ ys} + \text{length} \text{ xs}) \\ \text{(by comm\_sum)} & \quad =. 1 + \underline{(\text{length} \text{ xs} + \text{length} \text{ ys})} \end{aligned}$$

# Computation induction

## Example (cont.)

### Case 2

To show:  $\text{length} (\text{splice} (x:xs) \text{ ys})$   
           $=. \text{length} (x:xs) + \text{length} \text{ ys}$

IH:  $\text{length} (\text{splice} \text{ ys} \text{ xs})$   
      $=. \text{length} \text{ ys} + \text{length} \text{ xs}$

Proof

$$\begin{aligned} & \text{length} (\text{splice} (x:xs) \text{ ys}) \\ \text{(by def splice)} & \quad =. \text{length} (x : \text{splice} \text{ ys} \text{ xs}) \\ \text{(by def length)} & \quad =. 1 + \text{length} (\text{splice} \text{ ys} \text{ xs}) \\ \text{(by IH)} & \quad =. 1 + (\text{length} \text{ ys} + \text{length} \text{ xs}) \\ \text{(by comm\_sum)} & \quad =. 1 + (\text{length} \text{ xs} + \text{length} \text{ ys}) \\ \text{(by assoc\_sum)} & \quad =. \underline{(1 + \text{length} \text{ xs}) + \text{length} \text{ ys}} \end{aligned}$$

# Computation induction

## Example (cont.)

### Case 2

To show:  $\text{length} (\text{splice} (x:xs) \text{ ys})$   
           $=. \text{length} (x:xs) + \text{length} \text{ ys}$

IH:  $\text{length} (\text{splice} \text{ ys} \text{ xs})$   
      $=. \text{length} \text{ ys} + \text{length} \text{ xs}$

Proof

$$\begin{aligned} & \text{length} (\text{splice} (x:xs) \text{ ys}) \\ \text{(by def splice)} & \quad =. \text{length} (x : \text{splice} \text{ ys} \text{ xs}) \\ \text{(by def length)} & \quad =. 1 + \text{length} (\text{splice} \text{ ys} \text{ xs}) \\ \text{(by IH)} & \quad =. 1 + (\text{length} \text{ ys} + \text{length} \text{ xs}) \\ \text{(by comm\_sum)} & \quad =. 1 + (\text{length} \text{ xs} + \text{length} \text{ ys}) \\ \text{(by assoc\_sum)} & \quad =. (1 + \text{length} \text{ xs}) + \text{length} \text{ ys} \\ \text{(by def length)} & \quad =. \underline{\text{length} (x:xs)} + \text{length} \text{ ys} \end{aligned}$$

QED

QED



# Structural vs computation induction

- **structural induction**  
inductive proof over the structural definition of a datatype.
- **computation induction**  
inductive proof over the structural definition of a function.

# Plan

Correctness

# Correctness

How can we prove that two modules implement the same structure?

# Correctness

How can we prove that two modules implement the same structure?



How can we prove that the implementation of one module simulates its counterpart?

## Lists and sets

Each list  $[x_1, \dots, x_n]$  represents the set  $\{x_1, \dots, x_n\}$ .

# Lists and sets

Each list  $[x_1, \dots, x_n]$  represents the set  $\{x_1, \dots, x_n\}$ .  
In mathematical terms:

$$\alpha :: [a] \rightarrow \{a\}$$

$$\alpha [x_1, \dots, x_n] = \{x_1, \dots, x_n\}$$

$\alpha$  is an **abstraction function**.

# Lists and sets

Each list  $[x_1, \dots, x_n]$  represents the set  $\{x_1, \dots, x_n\}$ .  
In mathematical terms:

$$\alpha :: [a] \rightarrow \{a\}$$

$$\alpha [x_1, \dots, x_n] = \{x_1, \dots, x_n\}$$

$\alpha$  is an **abstraction function**.

Lists simulate sets  $\implies \alpha$  must be a homomorphism.

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs
```



## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs  
  
invar :: [a] -> Bool
```

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs  
  
invar :: [a] -> Bool  
invar [] = True
```

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs
```

```
invar :: [a] -> Bool  
invar [] = True  
invar (x:xs) = not (elem x xs) && invar xs
```

Simulation requirements:

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs
```

```
invar :: [a] -> Bool  
invar [] = True  
invar (x:xs) = not (elem x xs) && invar xs
```

Simulation requirements:

$$\alpha \text{ empty} = \emptyset$$

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs
```

```
invar :: [a] -> Bool  
invar [] = True  
invar (x:xs) = not (elem x xs) && invar xs
```

Simulation requirements:

$$\alpha \text{ empty} = \emptyset$$
$$\alpha \text{ invar xs} \implies \alpha (\text{insert } x \text{ xs}) = \{x\} \cup \alpha \text{ xs}$$

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs
```

```
invar :: [a] -> Bool  
invar [] = True  
invar (x:xs) = not (elem x xs) && invar xs
```

Simulation requirements:

$$\alpha \text{ empty} = \emptyset$$
$$\alpha \text{ invar xs} \implies \alpha (\text{insert } x \text{ xs}) = \{x\} \cup \alpha \text{ xs}$$
$$\alpha \text{ invar xs} \implies \text{isin } x \text{ xs} = x \in \alpha \text{ xs}$$

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs
```

```
invar :: [a] -> Bool  
invar [] = True  
invar (x:xs) = not (elem x xs) && invar xs
```

Simulation requirements:

$$\begin{aligned}\alpha \text{ empty} &= \emptyset \\ \alpha \text{ invar } xs &\implies \alpha (\text{insert } x \text{ } xs) = \{x\} \cup \alpha \text{ } xs \\ \alpha \text{ invar } xs &\implies \text{isin } x \text{ } xs = x \in \alpha \text{ } xs \\ \alpha \text{ invar } xs &\implies \text{size } xs = |\alpha \text{ } xs|\end{aligned}$$

## Lists and sets

```
empty = []  
insert x xs = if elem x xs then xs else x:xs  
isin x xs = elem x xs  
size xs = length xs
```

```
invar :: [a] -> Bool  
invar [] = True  
invar (x:xs) = not (elem x xs) && invar xs
```

Simulation requirements:

$$\alpha \text{ empty} = \emptyset$$
$$\alpha \text{ invar xs} \implies \alpha (\text{insert } x \text{ xs}) = \{x\} \cup \alpha \text{ xs}$$
$$\alpha \text{ invar xs} \implies \text{isin } x \text{ xs} = x \in \alpha \text{ xs}$$
$$\alpha \text{ invar xs} \implies \text{size xs} = |\alpha \text{ xs}|$$

invar must be preserved by every operation.



## Correctness proof strategy

Let  $C$  and  $A$  be two modules that have the same interface: a type  $T$  and a set of functions  $F$ .

## Correctness proof strategy

Let  $C$  and  $A$  be two modules that have the same interface: a type  $T$  and a set of functions  $F$ .

To prove that  $C$  is a correct implementation of  $A$  define

1. an **abstraction function**  $\alpha :: C.T \rightarrow A.T$
2. and an **invariant**  $\text{invar} :: C.T \rightarrow \text{Bool}$

and prove for each  $f \in F$ :

## Correctness proof strategy

Let  $C$  and  $A$  be two modules that have the same interface: a type  $T$  and a set of functions  $F$ .

To prove that  $C$  is a correct implementation of  $A$  define

1. an **abstraction function**  $\alpha :: C.T \rightarrow A.T$
2. and an **invariant**  $\text{invar} :: C.T \rightarrow \text{Bool}$

and prove for each  $f \in F$ :

- $\text{invar}$  is invariant

$$\text{invar } x_1 \wedge \dots \wedge \text{invar } x_n \implies \text{invar } (C.f \ x_1 \ \dots \ x_n)$$

## Correctness proof strategy

Let  $C$  and  $A$  be two modules that have the same interface: a type  $T$  and a set of functions  $F$ .

To prove that  $C$  is a correct implementation of  $A$  define

1. an **abstraction function**  $\alpha :: C.T \rightarrow A.T$
2. and an **invariant**  $\text{invar} :: C.T \rightarrow \text{Bool}$

and prove for each  $f \in F$ :

- $\text{invar}$  is invariant  
$$\text{invar } x_1 \wedge \dots \wedge \text{invar } x_n \implies \text{invar } (C.f \ x_1 \ \dots \ x_n)$$
- $C.f$  simulates  $A.f$   
$$\text{invar } x_1 \wedge \dots \wedge \text{invar } x_n \implies$$
$$\alpha \ (C.f \ x_1 \ \dots \ x_n) = A.f \ (\alpha \ x_1) \ \dots \ (\alpha \ x_n)$$

# Plan

## I/O

- I/O in Haskell

- Sequencing

- Interlude: Monads

## Side effects

Up until now we only considered programs that do not have side effects.

## Side effects

Up until now we only considered programs that do not have side effects.

To reason about programs like in mathematics, the programming language must have **referential transparency**. That is, any expression can be replaced by its value without changing the meaning of the program.

## Side effects

Up until now we only considered programs that do not have side effects.

To reason about programs like in mathematics, the programming language must have **referential transparency**. That is, any expression can be replaced by its value without changing the meaning of the program.

Programming languages that have referential transparency are called **pure**.



# I/O in Haskell

Haskell distinguishes expressions without side effects (**pure expressions**) from expressions with side effects (**actions**) by their type:

**`IO a`**

is the type of (I/O) actions that return a value of type `a`.

# I/O in Haskell

Haskell distinguishes expressions without side effects (**pure expressions**) from expressions with side effects (**actions**) by their type:

**`IO a`**

is the type of (I/O) actions that return a value of type `a`.

## Examples

- `Char`: the type of pure expressions returning a `Char`

# I/O in Haskell

Haskell distinguishes expressions without side effects (**pure expressions**) from expressions with side effects (**actions**) by their type:

**`IO a`**

is the type of (I/O) actions that return a value of type `a`.

## Examples

- `Char`: the type of pure expressions returning a `Char`
- `IO Char`: the type of actions returning a `Char`

# I/O in Haskell

Haskell distinguishes expressions without side effects (**pure expressions**) from expressions with side effects (**actions**) by their type:

**`IO a`**

is the type of (I/O) actions that return a value of type `a`.

## Examples

- `Char`: the type of pure expressions returning a `Char`
- `IO Char`: the type of actions returning a `Char`
- `IO ()`: the type of actions returning nothing

# I/O in Haskell

Haskell distinguishes expressions without side effects (**pure expressions**) from expressions with side effects (**actions**) by their type:

**`IO a`**

is the type of (I/O) actions that return a value of type `a`.

## Examples

- `Char`: the type of pure expressions returning a `Char`
- `IO Char`: the type of actions returning a `Char`
- `IO ()`: the type of actions returning nothing

**`()` is the type of empty tuples with the only value `()`.**

## Basic actions

- `getChar :: IO Char`  
Reads a `Char` from standard input,  
echoes it to standard output,  
and returns it as the result

## Basic actions

- `getChar :: IO Char`  
Reads a `Char` from standard input,  
echoes it to standard output,  
and returns it as the result
- `putChar :: Char -> IO ()`  
Writes a `Char` to standard output,  
and returns no result

# Basic actions

- `getChar :: IO Char`  
Reads a `Char` from standard input,  
echoes it to standard output,  
and returns it as the result
- `putChar :: Char -> IO ()`  
Writes a `Char` to standard output,  
and returns no result
- `return :: a -> IO a`  
Performs no action,  
just returns the given value as a result



# Read/Show

- Read: parsing String

```
class Read a where  
  read :: String -> a
```

# Read/Show

- Read: parsing String

```
class Read a where  
  read :: String -> a
```

- Show: converting to String

```
class Show a where  
  show :: a -> String
```

## Important actions

- `putStr :: String -> IO ()`  
Prints a string to standard output

## Important actions

- `putStr :: String -> IO ()`  
Prints a string to standard output
- `putStrLn :: String -> IO ()`  
Prints a string followed by a newline to standard output

## Important actions

- `putStr :: String -> IO ()`  
Prints a string to standard output
- `putStrLn :: String -> IO ()`  
Prints a string followed by a newline to standard output
- `getLine :: IO String`  
Reads everything up until a newline from standard input

# Sequencing

A sequence of actions can be combined into a single action with the keyword `do`.

# Sequencing

A sequence of actions can be combined into a single action with the keyword `do`.

## Example

```
get2 :: IO (Char,Char)
get2 = do x <- getChar    -- result is named x
         getChar          -- result is ignored
         y <- getChar
         return (x,y)
```

# Sequencing

General format:

do  $a_1$   
   $\vdots$   
   $a_n$



# Sequencing

General format:

do  $a_1$   
   $\vdots$   
   $a_n$

where each  $a_i$  can be one of

- an action  
  Effect: execute action
- $x \leftarrow action$   
  Effect: execute  $action :: IO\ a$ , give result the name  $x :: a$
- $let\ x = expr$   
  Effect: give  $expr$  the name  $x$

## Interlude: Monads

Monads are a general approach to computations that incur side effects.

## Interlude: Monads

Monads are a general approach to computations that incur side effects.

Idea: pipe data through the program implicitly.

## Interlude: Monads

Monads are a general approach to computations  
that incur side effects.

Idea: pipe data through the program implicitly.

In Haskell:

```
class Monad m where
```

## Interlude: Monads

Monads are a general approach to computations  
that incur side effects.

Idea: pipe data through the program implicitly.

In Haskell:

```
class Monad m where  
  (>>=) :: m a -> (a -> m b) -> m b
```

## Interlude: Monads

Monads are a general approach to computations  
that incur side effects.

Idea: pipe data through the program implicitly.

In Haskell:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

## Interlude: Monads

Monads are a general approach to computations  
that incur side effects.

Idea: pipe data through the program implicitly.

In Haskell:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

```
do x <- act1
  act2
```

is syntactic sugar for

```
act1 >>= (\x -> act2)
```

## Interlude: Monads

### Example: Maybe as a monad

```
instance Monad Maybe where
  m >=> f = case m of
    Nothing -> Nothing
    Just x   -> f x
  return v = Just v
```



## Interlude: Monads

### Example: Maybe as a monad

```
instance Monad Maybe where
  m >>= f = case m of
    Nothing -> Nothing
    Just x   -> f x
  return v = Just v
```

Using `do`, failure propagation and unwrapping of `Just` happens automatically.

## Interlude: Monads

### Example: Maybe as a monad

```
instance Monad Maybe where
  m >>= f = case m of
    Nothing -> Nothing
    Just x   -> f x
  return v = Just v
```

Using `do`, failure propagation and unwrapping of `Just` happens automatically.

```
x :: Maybe Int
y :: Maybe Int
sum2 :: Maybe Int
```

## Interlude: Monads

### Example: Maybe as a monad

```
instance Monad Maybe where
  m >>= f = case m of
    Nothing -> Nothing
    Just x   -> f x
  return v = Just v
```

Using `do`, failure propagation and unwrapping of `Just` happens automatically.

```
x :: Maybe Int
y :: Maybe Int
sum2 :: Maybe Int
sum2 = do
  a <- someMaybeInt
  b <- anotherMaybeInt
  return (a + b)
```

Plan

Evaluation

# Evaluation

Expressions are evaluated (**reduced**) by successively applying definitions until no further reduction is possible.

# Evaluation

Expressions are evaluated (**reduced**) by successively applying definitions until no further reduction is possible.

An expression may have many reducible sub-expressions:

sq (3+4)

A reducible expression is also called **redex**.

## Reduction strategies

- **innermost** reduces the innermost redex first

# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body



# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body
  - corresponds to **call by value**

# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body
  - corresponds to **call by value**
- **outermost** reduces the outermost redex first

# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body
  - corresponds to **call by value**
- **outermost** reduces the outermost redex first
  - unevaluated arguments are substituted into the function body

# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body
  - corresponds to **call by value**
- **outermost** reduces the outermost redex first
  - unevaluated arguments are substituted into the function body
  - corresponds to **call by name**

# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body
  - corresponds to **call by value**
- **outermost** reduces the outermost redex first
  - unevaluated arguments are substituted into the function body
  - corresponds to **call by name**
- **lazy** combines an outermost reduction strategy with the **sharing** of expressions.

# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body
  - corresponds to **call by value**
- **outermost** reduces the outermost redex first
  - unevaluated arguments are substituted into the function body
  - corresponds to **call by name**
- **lazy** combines an outermost reduction strategy with the **sharing** of expressions.
  - unevaluated arguments are substituted into the function body, but are only evaluated once for all copies of the same expression

# Reduction strategies

- **innermost** reduces the innermost redex first
  - arguments are evaluated before they are substituted into the function body
  - corresponds to **call by value**
- **outermost** reduces the outermost redex first
  - unevaluated arguments are substituted into the function body
  - corresponds to **call by name**
- **lazy** combines an outermost reduction strategy with the **sharing** of expressions.
  - unevaluated arguments are substituted into the function body, but are only evaluated once for all copies of the same expression
  - *call by need*

# Theorems

- Any two terminating evaluations of the same Haskell expression lead to the same final result.



# Theorems

- Any two terminating evaluations of the same Haskell expression lead to the same final result.
- If expression  $e$  has a terminating reduction sequence, then outermost reduction of  $e$  also terminates.

# Theorems

- Any two terminating evaluations of the same Haskell expression lead to the same final result.
- If expression  $e$  has a terminating reduction sequence, then outermost reduction of  $e$  also terminates.  
 $\implies$  outermost reduction terminates as often as possible

# Theorems

- Any two terminating evaluations of the same Haskell expression lead to the same final result.
- If expression  $e$  has a terminating reduction sequence, then outermost reduction of  $e$  also terminates.  
     $\implies$  outermost reduction terminates as often as possible
- Lazy evaluation never needs more steps than innermost reduction.

# Principles of lazy evaluation

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.

# Principles of lazy evaluation

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function.

# Principles of lazy evaluation

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function.
- Each argument is evaluated at most once. (sharing!)

# Principles of lazy evaluation

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function.
- Each argument is evaluated at most once. (sharing!)

Haskell never reduces inside a lambda

Why?

# Principles of lazy evaluation

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function.
- Each argument is evaluated at most once. (sharing!)

Haskell never reduces inside a lambda

Why?

- lazy evaluation uses as few steps as possible
- functions can only be applied



# Infinite lists

Example: head ones

```
ones :: [Int]
ones = 1 : ones
```

ones defines an infinite list of 1s. ones is called a **producer**.

# Infinite lists

Example: `head ones`

```
ones :: [Int]
ones = 1 : ones
```

`ones` defines an infinite list of 1s. `ones` is called a **producer**.

Outermost reduction:

`head ones`

# Infinite lists

Example: `head ones`

```
ones :: [Int]
ones = 1 : ones
```

`ones` defines an infinite list of 1s. `ones` is called a **producer**.

Outermost reduction:

```
head ones
= head (1 : ones)
```

# Infinite lists

Example: `head ones`

```
ones :: [Int]
ones = 1 : ones
```

`ones` defines an infinite list of 1s. `ones` is called a **producer**.

Outermost reduction:

```
head ones
= head (1 : ones)
= 1
```

Innermost reduction:

```
head ones
```

# Infinite lists

Example: `head ones`

```
ones :: [Int]
ones = 1 : ones
```

`ones` defines an infinite list of 1s. `ones` is called a **producer**.

Outermost reduction:

```
head ones
= head (1 : ones)
= 1
```

Innermost reduction:

```
head ones
= head (1 : ones)
```

# Infinite lists

Example: head ones

```
ones :: [Int]
ones = 1 : ones
```

ones defines an infinite list of 1s. ones is called a **producer**.

Outermost reduction:

```
head ones
= head (1 : ones)
= 1
```

Innermost reduction:

```
head ones
= head (1 : ones)
= head (1 : 1 : ones)
= ...
```

# Infinite lists

Haskell lists are never actually infinite  
but only potentially infinite

Lazy evaluation computes as much of the infinite list as needed

# Plan

Time complexity analysis



# Time complexity analysis

Assumption: One reduction step takes one time unit

# Time complexity analysis

Assumption: One reduction step takes one time unit

$T_f(n)$  = number of steps for the evaluation of  $f$  when applied to an argument of size  $n$  in the worst case

# Time complexity analysis

Assumption: One reduction step takes one time unit

$T_f(n)$  = number of steps for the evaluation of  $f$  when applied to an argument of size  $n$  in the worst case

Size is a specific measure based on the argument type of  $f$ .

# Time complexity analysis

Assumption: One reduction step takes one time unit

$T_f(n)$  = number of steps for the evaluation of  $f$  when applied to an argument of size  $n$  in the worst case

Size is a specific measure based on the argument type of  $f$ .

Calculating  $T_f(n)$ :

1. from the equations for  $f$  derive equations for  $T_f$
2. if the equations for  $T_f$  are recursive, solve them

# Time complexity analysis

## Example

```
[] ++ ys = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

# Time complexity analysis

## Example

```
[] ++ ys = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$

# Time complexity analysis

## Example

```
[] ++ ys = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$

$$T_{++}(m + 1, n) = T_{++}(m, n) + O(1)$$

# Time complexity analysis

## Example

$[] \mathrel{++} ys = ys$   
 $(x:xs) \mathrel{++} ys = x : (xs \mathrel{++} ys)$

$$T_{++}(0, n) = O(1)$$

$$T_{++}(m + 1, n) = T_{++}(m, n) + O(1)$$

$$\implies T_{++}(m, n) = O(m)$$