Functional Programming and Verification
revision course

Jonas Hübotter

June 13, 2020

Plan

Functional Programming and Haskell

- Basic Haskell
- Recursion, guards, pattern matching
- List comprehensions
- QuickCheck
- Polymorphism
- Currying, partial application, higher-order functions

Types

- Type aliases
- Type Classes
- Algebraic Data Types
- Modules, Abstract Data Types
- Type inference

Correctness

I/O

Lazy evaluation

Complexity and optimization

Basic Haskell

function types	`f :: a -> b -> c`
function definitions	`f a b = a + b`
function application	`f 1 2`
function composition	`f . g` means $f(g(x))$
conditional	`if True then a else b`
prefix/infix precedence	``f a `g` b`` means ``(f a) `g` b``
$ sign	``f $ a `g` b`` means ``f (a `g` b)``

Types

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

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

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

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

Intuitively: `(:) :: a -> [a] -> [a]`.

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
`take :: Int -> [a] -> [a]`	prefix of list with given length
`drop :: Int -> [a] -> [a]`	suffix of list 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

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]`	suffix of list 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

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 =` e_1 `in` e_2

defines `x` locally in e_2.

Local definitions

`let x =` e_1 `in` e_2

defines `x` locally in e_2.

e_2 `where x =` e_1

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

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 0
  where
    aux :: Integer -> Integer -> Integer
    aux n acc
```

Accumulating parameter

Alternatively, `factorial` could be defined as

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

Accumulating parameter

Alternatively, `factorial` could be defined as

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

```haskell
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 or type?

`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 or type?

```
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 or type?

```
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 or type?

```
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 or type?

```
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

$$[\ \mathit{expr}\ \mid\ E_1,\ \ldots,\ E_n\ ]$$

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

- a generator is of the form *pattern* <- *listexpression*
- 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 | x <- [a1,...,an]]
= (let x = a1 in [e]) ++ · · · ++ (let x = an in [e])

[e | b]
```

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]
```

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. 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 =
  max2 x y = max 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 =
  max2 x (max2 y z) = max2 (max2 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 =
  n > 2 ==> 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 > 2 ==> 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 -> a -> a
```

Function `(+)` has type `a -> a -> a` for any type `a` of the type class `Num`.

Type constraints

Type variables can be constrained by type constraints.

```
(+) :: Num a => a -> a -> a
```

Function `(+)` has type `a -> a -> a` for any type `a` of the type class `Num`.

Some type classes:

1. `Num`
2. `Integral`
3. `Fractional`
4. `Ord`
5. `Eq`
6. `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, partial application, higher-order functions

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

Currying, partial application, higher-order functions

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

Example

```
f :: Int -> Int -> Int
f x y = x + y

f a b
= a + b
```

```
f :: Int -> (Int -> Int)
f x = \y -> x + y

(f a) b
= (\y -> a + y) b
= a + b
```

Currying, partial application, higher-order functions

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

Example

```
f :: Int -> Int -> Int
f x y = x + y

f a b
= a + b
```

```
f :: Int -> (Int -> Int)
f x = \y -> x + y

(f a) b
= (\y -> 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 abstractionanonymous function) is a function without a name.

Examples

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

Anonymous functions (lambdas)

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

Examples

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

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

Anonymous functions (lambdas)

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

Examples

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

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

Anonymous functions (lambdas)

An anonymous function (or lambda abstractionanonymous function) 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

```
i :: Integer -> (a -> a) -> (a -> a)
succ :: Integer -> Integer
```

Anonymous functions (lambdas)

An anonymous function (or lambda abstractionanonymous function) 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

```
i :: 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.
A function is partially applied when some arguments have already been applied to a function (that is 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 (that is 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 (that is 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 (that is 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

```
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

```
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

```
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

```
filter :: (a -> Bool) -> [a] -> [a]

map :: (a -> b) -> [a] -> [b]

all, any :: (a -> Bool) -> [a] -> Bool

takeWhile, dropWhile :: (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

```
filter :: (a -> Bool) -> [a] -> [a]

map :: (a -> b) -> [a] -> [b]

all, any :: (a -> Bool) -> [a] -> Bool

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]

(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

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

Functional Programming and Haskell
- Basic Haskell
- Recursion, guards, pattern matching
- List comprehensions
- QuickCheck
- Polymorphism
- Currying, partial application, higher-order functions

Types
- Type aliases
- Type Classes
- Algebraic Data Types
- Modules, Abstract Data Types
- Type inference

Correctness

I/O

Lazy evaluation

Complexity and optimization

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.

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:

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
  (==) :: a -> a -> Bool
```

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.

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
```

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
```

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

- data types with a custom shape
- defines a type along with constructors to build values of that type

Algebraic Data Types

- data types with a custom shape
- defines a type along with constructors to build values of that type

$$\texttt{data}\ \mathit{type}\ a_1 \ldots a_n\ \texttt{=}\ \mathit{constructor}\ \ a_1 \ldots a_n\ \ \texttt{|}\ \ \texttt{...}$$

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)
```

Constructors vs Types (again)

Repetition:

- constructors can be used in expressions to build values of a type.
- types can be used in type signatures to hint at the type of bindings.

Constructors vs Types (again)

Repetition:

- constructors can be used in expressions to build values of a type.
- types can be used in type signatures to hint at the type of bindings.

An algebraic data types is a custom type with one or more constructors.

Constructors vs Types (again)

Constructors can have varying arity:

Constructors vs Types (again)

Constructors can have varying arity:

- a constructor with arity 0 acts like a value of the algebraic data type

Constructors vs Types (again)

Constructors can have varying arity:

- a constructor with arity 0 acts like a value of the algebraic data type
- a constructor with arity k combines k values of different types into a single value of the algebraic data type

Constructors vs Types (again)

Constructors can have varying arity:

- a constructor with arity 0 acts like a value of the algebraic data type
- a constructor with arity k combines k values of different types into a single value of the algebraic data type

Constructors are functions that unambiguously construct the value of a type.

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 -> 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 -> 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 -> 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 -> 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 -> 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 -> 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, Abstract Data Types

Modules

Collection of type, function, class and other definitions.

Modules, Abstract Data Types

Modules

Collection of type, function, class and other definitions.

Examples

`module M where`

exports everything defined in `M`

Modules, Abstract Data Types

Modules

Collection of type, function, class and other definitions.

Examples

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

`module M (T, f, ...) where`
exports everything defined in `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 witha custom shape)

`type` vs `data` vs `newtype`

- `type` is used to create type aliases
- `data` is used to create algebraic data types (types witha custom shape)
- `newtypre` is used to create a custom constructor for a single type

`type` vs `data` vs `newtype`

- `type` is used to create type aliases
- `data` is used to create algebraic data types (types witha custom shape)
- `newtypre` is used to create a custom constructor for a single type
  - syntax "subset" of the syntax for `data`
  - may only have a *single* constructor taking a *single* argument
  - introduces no runtime overhead
  - creates *strict* types while `data` creates *lazy* types

Type inference

How to infer/reconstruct the type of an expression.

Type inference

How to infer/reconstruct the type of an expression.

Given an expression e.

Type inference

How to infer/reconstruct the type of an expression.

Given an expression e.

1. give all variables in e distinct type variables

Type inference

How to infer/reconstruct 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

How to infer/reconstruct 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

How to infer/reconstruct 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 (head u) (last (concat v))`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 1

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 1

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

Step 2

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 1

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

Step 2

1. `head :: [c] -> c`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 1

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

Step 2

1. `head :: [c] -> c`
2. `concat :: [[d]] -> [d]`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 1

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

Step 2

1. `head :: [c] -> c`
2. `concat :: [[d]] -> [d]`
3. `last :: [e] -> e`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 1

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

Step 2

1. `head :: [c] -> c`
2. `concat :: [[d]] -> [d]`
3. `last :: [e] -> e`
4. `min :: Ord f => f -> f -> f`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 3

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 3

1. from `head u` derive `[c] = a`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 3

1. from `head u` derive `[c] = a`
2. from `concat v` derive `[[d]] = b`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 3

1. from `head u` derive `[c] = a`
2. from `concat v` derive `[[d]] = b`
3. from `last (concat v)` derive `[e] = [d]`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Step 3

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

Type inference

Example

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> f`

Step 4

Type inference

Example

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> f`

Step 4

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

Type inference

Example

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> f`

Step 4

1. apply `[c] = a` and update
   - `u :: [c]`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> f`

Step 4

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

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> f`

Step 4

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

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> 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

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> 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

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> 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]]`
   - `concat :: [[e]] -> [e]`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> 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]]`
   - `concat :: [[e]] -> [e]`
4. apply `f = c` and update

Type inference

Example

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> 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]]`
   - `concat :: [[e]] -> [e]`
4. apply `f = c` and update
   - `u :: [f]`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`
Goal `f :: a -> b -> 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]]`
   - `concat :: [[e]] -> [e]`
4. apply `f = c` and update
   - `u :: [f]`
   - `head :: [f] -> f`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> f`

Step 4 (cont.)

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> f`

Step 4 (cont.)

1. apply `f = e` and update

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> f`

Step 4 (cont.)

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

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> f`

Step 4 (cont.)

1. apply `f = e` and update
   - `v :: [[f]]`
   - `concat :: [[f]] -> [f]`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> f`

Step 4 (cont.)

1. apply `f = e` and update
   - `v :: [[f]]`
   - `concat :: [[f]] -> [f]`
   - `last :: [[f]] -> [f]`

Type inference

Example

Given `f u v = min (head u) (last (concat v))`

Goal `f :: a -> b -> f`

Step 4 (cont.)

1. apply `f = e` and update
   - `v :: [[f]]`
   - `concat :: [[f]] -> [f]`
   - `last :: [[f]] -> [f]`
2. no further simplification possible,
   return `f :: Ord f => [f] -> [[f]] -> f`

Plan

- Functional Programming and Haskell
  - Basic Haskell
  - Recursion, guards, pattern matching
  - List comprehensions
  - QuickCheck
  - Polymorphism
  - Currying, partial application, higher-order functions
- Types
  - Type aliases
  - Type Classes
  - Algebraic Data Types
  - Modules, Abstract Data Types
  - Type inference
- **Correctness**
- I/O
- Lazy evaluation
- Complexity and optimization

Correctness

How can we prove that two modules implement the same structure?

Correctness

How can we prove that two modules implement the same structure?

$$\Longleftrightarrow$$

How can we prove that the implementation of one module simulates its counterpart?

Lists and sets

Each list `[x1, ..., xn]` represents the set $\{x_1, \ldots, x_n\}$.

Lists and sets

Each list $[x_1\,,\ \ldots,\ x_n]$ represents the set $\{x_1, \ldots, x_n\}$.
In mathematical terms:

α `:: [a] ->` $\{a\}$
α `[`x_1`, ..., `x_n`] = `$\{x$_1, $\ldots$, x_$n\}$

α is an abstraction function.

Lists and sets

Each list `[`x_1`, ..., `x_n`]` represents the set $\{x_1, \ldots, x_n\}$.
In mathematical terms:

α `:: [a] ->` $\{a\}$
α `[`x`_1, ..., `x`_`n`] =` $\{$`x_1, ..., x_n`$\}$

α is an abstraction function.

Lists simulate sets $\Longrightarrow$ α 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:

α `empty` = $\{\}$

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:

α `empty` = $\{\}$

`invar xs` $\Longrightarrow$ α `(insert x xs)` = $\{$`x`$\} \cup \alpha$ `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:

α `empty` = $\{\}$

`invar xs` $\Longrightarrow$ α `(insert x xs)` = $\{$`x`$\}$ $\cup$ α `xs`

`invar xs` $\Longrightarrow$ `isin x xs` = `x` $\in$ α `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:

$\alpha\ \texttt{empty} = \{\}$

$\texttt{invar xs} \implies \alpha\ (\texttt{insert x xs}) = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$

$\texttt{invar xs} \implies \texttt{isin x xs} = \texttt{x} \in \alpha\ \texttt{xs}$

$\texttt{invar xs} \implies \texttt{size xs} = |\alpha\ \texttt{xs}|$

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:

$\alpha\ \texttt{empty} = \{\}$

$\texttt{invar xs} \implies \alpha\ \texttt{(insert x xs)} = \{\texttt{x}\} \cup \alpha\ \texttt{xs}$

$\texttt{invar xs} \implies \texttt{isin x xs} = \texttt{x} \in \alpha\ \texttt{xs}$

$\texttt{invar xs} \implies \texttt{size xs} = |\alpha\ \texttt{xs}|$

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 α `::` $C.T$ `->` $A.T$
2. and an invariant `invar ::` $C.T$ `-> 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 α `::` $C.T$ `->` $A.T$
2. and an invariant `invar` `::` $C.T$ `-> Bool`

and prove for each $f \in F$:

- `invar` is invariant

  $\texttt{invar}\ x_1 \wedge \cdots \wedge \texttt{invar}\ x_n \implies \texttt{invar}\ (C.f\ x_1\ \ldots\ x_n)$

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 α `::` $C.T$ `->` $A.T$
2. and an invariant `invar ::` $C.T$ `-> Bool`

and prove for each $f \in F$:

- `invar` is invariant
  `invar` $x_1 \wedge \cdots \wedge$ `invar` $x_n \implies$ `invar` $(C.f\ x_1\ \ldots\ x_n)$
- $C.f$ simulates $A.f$
  `invar` $x_1 \wedge \cdots \wedge$ `invar` $x_n \implies$
  $\alpha\ (C.f\ x_1\ \ldots\ x_n) = A.f\ (\alpha\ x_1)\ \ldots\ (\alpha\ x_n)$

Plan

Functional Programming and Haskell
- Basic Haskell
- Recursion, guards, pattern matching
- List comprehensions
- QuickCheck
- Polymorphism
- Currying, partial application, higher-order functions

Types
- Type aliases
- Type Classes
- Algebraic Data Types
- Modules, Abstract Data Types
- Type inference

Correctness

I/O

Lazy evaluation

Complexity and optimization

I/O

Side effects

Up until now we only considered programs that do not have side effects.

I/O

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.

I/O

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

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:

$$\begin{array}{ll}\texttt{do} & a_1 \\ & \vdots \\ & a_n\end{array}$$

Sequencing

General format:

$$\begin{array}{l}\texttt{do}\ a_1\\ \quad\ \vdots\\ \quad\ a_n\end{array}$$

where each a_i can be one of

- an action
  Effect: execute action
- x `<-` *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)
```

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 ()`	print string to standard output
`putStrLn :: String -> IO ()`	print string followed by newline to standard output
`getLine :: IO String`	read everything up until newline from standard input

Plan

Functional Programming and Haskell
- Basic Haskell
- Recursion, guards, pattern matching
- List comprehensions
- QuickCheck
- Polymorphism
- Currying, partial application, higher-order functions

Types
- Type aliases
- Type Classes
- Algebraic Data Types
- Modules, Abstract Data Types
- Type inference

Correctness

I/O

Lazy evaluation

Complexity and optimization

Lazy evaluation

Expressions are evaluated (reduced) by successively applying definitions until no further reduction is possible.

Lazy evaluation

Expressions are evaluated (reduced) by successively applying definitions until no further reduction is possible.

An expression may have many reducible sub-expressions:

$$\underline{\texttt{sq}\ \underline{\texttt{(3+4)}}}$$

A reducible expression is also called redex.

Reduction strategies

Innermost

Reduces innermost redex first.

Reduction strategies

Innermost

Reduces innermost redex first.

$\Longrightarrow$ arguments are evaluated before they are substituted into the function body.

Reduction strategies

Innermost

Reduces innermost redex first.

$\Longrightarrow$ arguments are evaluated before they are substituted into the function body.

$\Longrightarrow$ corresponds to call by value

Reduction strategies

Innermost

Reduces innermost redex first.

$\implies$ arguments are evaluated before they are substituted into the function body.

$\implies$ corresponds to call by value

Outermost

Reduces outermost redex first.

Reduction strategies

Innermost

Reduces innermost redex first.

$\Longrightarrow$ arguments are evaluated before they are substituted into the function body.

$\Longrightarrow$ corresponds to call by value

Outermost

Reduces outermost redex first.

$\Longrightarrow$ unevaluated arguments are substituted into the function body.

Reduction strategies

Innermost

Reduces innermost redex first.

$\implies$ arguments are evaluated before they are substituted into the function body.

$\implies$ corresponds to call by value

Outermost

Reduces outermost redex first.

$\implies$ unevaluated arguments are substituted into the function body.

$\implies$ corresponds to call by name

Reduction strategies

Innermost

Reduces innermost redex first.

$\Longrightarrow$ arguments are evaluated before they are substituted into the function body.

$\Longrightarrow$ corresponds to call by value

Outermost

Reduces outermost redex first.

$\Longrightarrow$ unevaluated arguments are substituted into the function body.

$\Longrightarrow$ corresponds to call by name

Lazy

Combines an outermost reduction strategy with the sharing of expressions.

Reduction strategies

Innermost

Reduces innermost redex first.

$\implies$ arguments are evaluated before they are substituted into the function body.

$\implies$ corresponds to call by value

Outermost

Reduces outermost redex first.

$\implies$ unevaluated arguments are substituted into the function body.

$\implies$ corresponds to call by name

Lazy

Combines an outermost reduction strategy with the sharing of expressions.

$\implies$ unevaluated arguments are substituted into the function body, but are only evaluated once for all copies of the same expression.

Reduction strategies

Innermost

Reduces innermost redex first.

$\Longrightarrow$ arguments are evaluated before they are substituted into the function body.

$\Longrightarrow$ corresponds to call by value

Outermost

Reduces outermost redex first.

$\Longrightarrow$ unevaluated arguments are substituted into the function body.

$\Longrightarrow$ corresponds to call by name

Lazy

Combines an outermost reduction strategy with the sharing of expressions.

$\Longrightarrow$ unevaluated arguments are substituted into the function body, but are only evaluated once for all copies of the same expression.

$\Longrightarrow$ *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.

  $\Longrightarrow$ 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.
  $\Longrightarrow$ 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.
- 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

Functional Programming and Haskell
- Basic Haskell
- Recursion, guards, pattern matching
- List comprehensions
- QuickCheck
- Polymorphism
- Currying, partial application, higher-order functions

Types
- Type aliases
- Type Classes
- Algebraic Data Types
- Modules, Abstract Data Types
- Type inference

Correctness

I/O

Lazy evaluation

Complexity and optimization

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

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$
$$T_{++}(m + 1, n) = T_{++}(m, n) + O(1)$$
$$\implies T_{++}(m, n) = O(m)$$

Optimization

- no duplication (use `where` or `let`)

Optimization

- no duplication (use `where` or `let`)
- use tail recursion

Optimization

- no duplication (use `where` or `let`)
- use tail recursion
- avoid multiple traversals of the same data structure

Optimization

- no duplication (use `where` or `let`)
- use tail recursion
- avoid multiple traversals of the same data structure
- avoid intermediate data structures

Optimization

- no duplication (use `where` or `let`)
- use tail recursion
- avoid multiple traversals of the same data structure
- avoid intermediate data structures
- pre-compute expensive operations