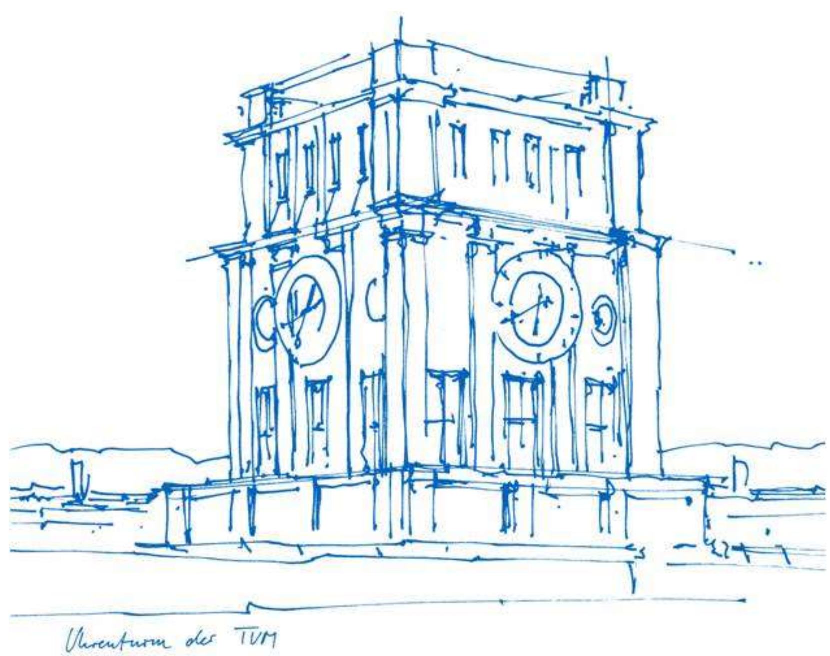
Revision course
Functional Programming and Verification

Jonas Hübotter



1. Functional Programming and Haskell

Basic Haskell

Recursion, guards, pattern matching

List comprehensions

QuickCheck

Polymorphism

Currying, partial application

1.1 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.

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

Lists (2) – 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 lists of components

Lists (3) – Prelude functions

`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/

Lists (4) – 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.

e_2 `where x =` e_1

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

1.2 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
  | n == 0 = 1                          -- base case
  | n > 0  = n * factorial (n - 1)      -- recursive case
```

Recursion (2) – 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)
```

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

Pattern matching (2) - 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.

Constructor or type?

```
False
(:)
Maybe
Just
Nothing
```

Pattern matching (3) - case

Pattern matching in nested expressions

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

1.3 List comprehensions

$$[\ expr \mid E_1, \ldots, E_n\ ]$$

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

- a generator is of the form `pattern <- list expression`
- a test is a Boolean expression

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.

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]]
= (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 []
```

1.4 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.

1.5 Polymorphism

Idea: one function definition, having many types.

`length :: [a] -> Int` is defined for all types `a`.

`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.

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`.

Some type classes:

- `Num`
- `Integral`
- `Fractional`
- `Ord`
- `Eq`
- `Show`

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)]]
```

1.6 Currying, partial application

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.

- function type signatures are right associative
- function application is left associative

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 `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.

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.

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)
```

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
```

2. Types

Type aliases

Type Classes

Algebraic Data Types

Modules, Abstract Data Types

Type inference

2.1 Type aliases

Allows the renaming of a more comples type expression.

Examples

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

2.2 Type Classes

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

Similar concepts are commonly called *interfaces*.

1. define set of functions (~ creating an interface)
2. implement set of functions for members of type class (~ implementing an interface)

Example

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

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

Constrained instances

Instances 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`.

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)
```

2.3 Algebraic Data Types

Custom data that allow us to specify the shape of each element.

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.

An Algebraic Data Type is a custom type with one or more constructors.

Constructors can have varying arity:

- a constructor with arity `0` acts like a value of the type
- a constructor with arity `k` combines `k` values of varying types into a single value of the type

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
```

2.4 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 only `T, f, …`

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 can have the same name as a type.

Abstract Data Types

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

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

`data` is used to create custom data types

`newtype` is used to create a custom constructor for a single type

- syntax similar to `data`
- may only have a single constructor taking a single argument
- introduces no runtime overhead
- creates *strict* types while `data` creates *lazy* types

2.5 Type inference

How to infer/reconstruct the type of an expression.

Algorithm (sketch)

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 that does not use type variables from (1)
3. for each sub-expression in e set up an equation linking parameters and arguments
4. simplify set of equations by replacing equivalences and constructors

Type inference example

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

Step 1
- `u :: a`
- `v :: b`

Step 2
- `head :: [c] -> c`
- `concat :: [[d]] -> [d]`
- `last :: [e] -> e`
- `min :: Ord f => f -> f -> f`

Step 3
- from `head u` derive `[c] = a`
- from `concat v` derive `[[d]] = b`
- from `last (concat v)` derive `[e] = [d]`
- from `min (head u) (last (concat v))` derive `f = c` and `f = e`

Type inference example (2)

Step 4

- goal `f :: a -> b -> f`
- apply `[c] = a` and update
  - `u :: [c]`
- apply `[[d]] = b` and update
  - `v :: [[d]]`
- apply `[e] = [d]` to get `e = d` and update
  - `v :: [[e]]`
  - `concat :: [[e]] -> [e]`
- apply `c = f` and update
  - `u :: [f]`
  - `head :: [f] -> f`
- apply `e = f` and update
  - `v :: [[f]]`
  - `concat :: [[f]] -> [f]`
  - `last :: [[f]] -> [f]`
- no further simplification possible, return `f :: Ord f => [f] -> [[f]] -> f`