Programming Paradigms
Unit 12 — Functions and Data Types in Haskell

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE
Outline

1. Functions
2. User-Defined Data Types
3. Type Classes Revisited
Outline

1 Functions

2 User-Defined Data Types

3 Type Classes Revisited
Now that we have modules, let’s write slightly more sophisticated functions. Haskell does pattern matching like Prolog.

- When you call a function, Haskell goes from top to bottom to find a signature (i.e., pattern) that matches the call.
- The order of the function definitions matters.

Different from Prolog:
- Only one function definition is executed (i.e., no backtracking!)

The following function computes the factorial of a number:

```haskell
module Factorial (factorial) where

factorial :: Integer -> Integer
factorial 0 = 1
factorial x = x * factorial (x-1)
```
Pattern Matching and Guards

- If you need to match in a different or particular order, you can use guards.
- Guards are boolean conditions that constrain the argument, and hence the pattern matching process.
- Guards are indicated by pipes | that follow a function’s name and its parameters.
- If the guard is satisfied, the corresponding function body is executed.
- Otherwise, pattern matching jumps to the next guard.

```haskell
module FactorialGuards (
  factorial
) where

factorial :: Integer -> Integer
factorial x
  | x > 1 = x * factorial (x-1)
  | otherwise = 1

Often, the last guard is otherwise, which catches everything.
Usually, a blank or tab is required before each | symbol.
```
We are now going to unleash more of the power of Haskell

Let's write a function for the Fibonacci numbers using lazy evaluation

Lazy evaluation means that expressions are not evaluated when they are bound to variables, but when their results are needed by other computations.

It is often used in combination with list construction to construct an infinite list, which however never need to be computed completely.

```haskell
module Fibonacci (
    lazyFib,
    fib
  ) where

lazyFib :: Integer -> Integer -> [Integer]
lazyFib x y = x:(lazyFib y (x + y))

fib :: Int -> Integer
fib x = head(drop (x-1) (lazyFib 1 1))
```
Lazy Evaluation of Functions/2

- lazyFib generates an infinite sequence of Fibonacci numbers

  > lazyFib 1 1
  [1,1,2,3,5,8,13,21,34,55,89,144,...

- Due to lazy evaluation, we never actually generate the whole list

- fib drops the first \( x-1 \) elements of the “infinite” list of Fibonacci numbers, and then takes the head of the remaining list

  > fib 4
  3
Combining lots of functions to get a result is a common pattern in functional languages.

This is called **function composition**.

As this is very common, Haskell has a shortcut notation.

Instead of writing

\[ f(g(h(i(j(k(l(m(n(o(x)))))))))) \]

you can write

\[ f \cdot g \cdot h \cdot i \cdot j \cdot k \cdot l \cdot m \cdot n \cdot o \cdot x \]
So our Fibonacci code could be rewritten into

```
module Fibonacci (
  lazyFib,
  fib
) where

lazyFib :: Integer -> Integer -> [Integer]
lazyFib x y = x:(lazyFib y (x + y))

fib :: Int -> Integer
fib x = (head.drop (x-1)) (lazyFib 1 1)
```
Anonymous Functions or Lambdas

- **Anonymous functions** are termed **lambdas** in Haskell and do not have a name.
- They are useful if a function is needed only once.
  - Usually used to pass a function as parameter to a higher-order function.
- The syntax is:

  \[
  (\text{parameter}_1, \ldots, \text{parameter}_n \rightarrow \text{function body})
  \]

- Let's write a function that just returns the input parameter:

  \[
  (> (\text{x} \rightarrow \text{x}) "mirror, mirror on the wall"
  "mirror, mirror on the wall"
  (> (\text{x} \rightarrow \text{x} + " world!") "Hello"
  "Hello world!"
  )
  \]
Haskell (as functional language) supports higher-order functions, i.e., functions that can take functions as parameters or return functions.

Examples of built-in higher-order functions are the usual list functions, such as `map`, `foldl`, `foldr`, `filter`.

- `map` expects
  - a function and a list as input and
  - returns a list which is the result of applying the function to each element in the input list.

```haskell
> map (\x -> x * x) [1,2,3]
[1,4,9]
> map (+ 1) [1,2,3]
[2,3,4]
```
Higher-Order Functions/2

- **foldl** expects
  - as input a function with two input parameter, an initial accumulator value, and an input list
  - and returns a single value resulting from applying the function to each element in the list and the accumulator

```haskell
> foldl (\x sum -> sum + x) 0 [1..10]
55
```

```haskell
> foldl (+) 0 [1,2,3]
6
```
Every function in Haskell officially only takes one parameter.

We’ve already defined functions with multiple input parameters, so how does this work?

Haskell uses the concept of curried functions:
- A function with multiple arguments is split into multiple functions with one argument each.
- That is, functions are applied partially, i.e., one parameter at a time.

Let’s have a look at an example.
Consider a function to multiply two numbers

> let prod x y = x * y

What is really going on behind the scences, if Haskell computes the product of two numbers, say prod 2 4?

1. Apply prod 2, which returns the function \( \lambda y \rightarrow 2 \times y \)
2. Apply \( \lambda y \rightarrow 2 \times y \) 4, which gives \( 2 \times 4 \), yielding the final result 8

So what is actually computed is

\( (\text{prod } 2) \ 4 \)

\( (\text{prod } 2) \) is a partial evaluation of a function, i.e.,

- only one argument is provided and substituted in the function definition
- the partially evaluated function is returned
Let’s have a look at the type of the function `prod`

```haskell
> :t prod
prod :: Num a => a -> a -> a
```

What this really says is the following:
- `prod` takes an input parameter of type `a` and returns a function that takes an input parameter of type `a` and returns a value of type `a`.

To make this more explicit, it could be written as

```haskell
Num a => a -> (a -> a)
```

... and the function can also be called as

```haskell
> (prod 2) 4
8
```
Advantages of Curried Functions

- We can create new functions on the fly, already partially evaluating a function in a different context.
- It makes formal proofs about programs simpler, because all functions are treated in the same way.
- There are some techniques used in Haskell where currying becomes important.
Partial Application of Functions

- Partial application of functions binds some of the arguments but not all and returns a function that is partially evaluated.

- Consider again the function `prod` to multiply two numbers:

  ```haskell
  > let prod x y = x * y
  ```

- We can partially apply `prod` to create some new functions:

  ```haskell
  > let double = prod 2
  > let triple = prod 3
  ```

- These two function definitions apply `prod`, but only with one parameter:
  - This substitutes the first parameter in the definition of `prod` and returns a partially evaluated function, e.g., `prod 2` gives `prod y = 2 * y`.

- The newly defined functions work just as you expect:

  ```haskell
  > double 3
  6
  > triple 4
  12
  ```
Outline

1 Functions

2 User-Defined Data Types

3 Type Classes Revisited
User-Defined Types

- You can declare your own data types using the keyword `data`
- The simplest version is an enumeration: a finite list of values separated by a vertical bar (|)

```haskell
data Verdict = Guilty | Innocent
```

- That means, a variable of type `Verdict` will have a single value, either Guilty or Innocent
- `Verdict` is called a type constructor
- The parts after the `=` are called value constructors, as they specify the different values that this type can have
**Enumerated Types/1**

- In the following module definition, Suit and Rank are **type constructors**

```haskell
module Cards where

data Suit = Spades | Clubs | Hearts | Diamonds

data Rank = Ace | Ten | King | Queen | Jack

```

- Loading this module and then trying to use one of these values leads to an error message

```haskell
> :l Cards
[1 of 1] Compiling Cards
Ok, modules loaded: Cards.

*Cards> Spades
<interactive>:1:1:
    No instance for (Show Suit)
...
```
Enumerated Types/2

- Haskell tells us that it does not know how to show values of these types.
- In order to show them, we have to make `Suit` and `Rank` instances of the type class `Show` using the keyword `deriving`.

```haskell
module Cards where

data Suit = Spades | Clubs | Hearts | Diamonds
  deriving (Show)
data Rank = Ace | Ten | King | Queen | Jack
  deriving (Show)
```

- Now we can load the module again and show the values.

  ```haskell
  > Clubs
  Clubs
  > Ten
  Ten
  ```
When building more complex composite types, we can use alias types, which start with the keyword `type`.

```haskell
data Suit = Spades | Clubs | Hearts | Diamonds
  deriving (Show)
data Rank = Ace | Ten | King | Queen | Jack
  deriving (Show)
type Card = (Rank,Suit)
type Hand = [Card]
```

> let card = (Ten,Hearts)
> card
(Ten,Hearts)

Card is now essentially a synonym (alias type) for (Rank,Suit), and Hand for [Card].

Type synonyms are mostly just a convenience.
An alternative way is to use a new type constructor (keyword `data`)

```haskell
data Suit = Spades | Clubs | Hearts | Diamonds
  deriving (Show)
data Rank = Ace | Ten | King | Queen | Jack
  deriving (Show)
data Card = Crd(Rank,Suit) deriving (Show)
data Hand = Hnd[Card] deriving (Show)

> let card = Crd(Ten,Hearts)
> card
Crd (Ten,Hearts)

> let hand = Hnd[Crd(Ten,Hearts), Crd(King,Diamonds)]
> hand
Hnd [Crd (Ten,Hearts), Crd (King,Diamonds)]
```
If we want to know the value of a card, we could write a function taking a `Rank` and returning an `Int`:

```haskell
value :: Rank -> Int
value Ace = 11
value Ten = 10
value King = 4
value Queen = 3
value Jack = 2
```

Applying this function:

```haskell
> let card = (Ace,Spades)
> let (r,s) = card
> value r
11
```
Value Constructors and Optional Parameters

- Value constructors can optionally be followed by some types (parameters) that define the values it will contain.
- Let's define a type to store shapes, such as circles or rectangles:

```haskell
data Shape = Circle Float Float Float | Rectangle Float Float Float Float deriving(Show)
```

```haskell
> let c = Circle 10 10 5
> c
Circle 10.0 10.0 5.0
```

- Circle and Rectangle are value constructors followed by type parameters:
  - Circle: the first two values are the center and the third value is the radius
  - Rectangle: upper-left corner and lower-right corner
Value constructors are actually functions like (almost) everything else in Haskell; they ultimately return a value of a data type.

Let's take a look at the type signatures for the two value constructors of the Shape data type:

```
> :t Circle
Circle :: Float -> Float -> Float -> Shape

> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

Both value constructors take Float parameters in input and return a Shape.
Using User-Defined Data Types

- Lets write a function to compute the surface of the shapes

```haskell
module Surface (surface) where

surface :: Shape -> Float
surface (Circle _ _ r) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs (x2 - x1)) * (abs (y2 - y1))
```

```haskell
> surface (Circle 10 10 5)
78.53982

> surface (Rectangle 0 0 10 10)
78.53982
```

- The underscore (_) means that this parameter is not used (as in Prolog)

- Notice that the value constructors Circle and Rectangle are used in pattern matching
Polymorphism in Functions

A function that reverses a list of cards could look like this

```haskell
backwards :: Hand -> Hand
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

However, that would restrict the function to lists of items of type `Hand`

If we want it to work with general lists, we can introduce any type by using `type variables`

```haskell
backwards :: [a] -> [a]
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

This is known as **polymorphism**, as `a` can be any type

`backwards` takes now a list of elements of type `a` and produces a list of elements of the same type `a`

- backwards is **polymorphic**
Polymorphism in User-Defined Types/1

- User-defined types can also be made **polymorphic** by using so-called **type variables**

- For example, you need a type that stores a list of pairs of **any type**

```haskell
data ListOfPairs a = LoP [(a,a)] deriving (Show)
```

```haskell
> let list1 = LoP[(1,2),(2,3),(3,4)]
> list1
LoP [(1,2),(2,3),(3,4)]
> let list2 = LoP[('a','b'),('b','c'),('c','d')]
> list2
LoP [('a','b'),('b','c'),('c','d')]
```

- Notice the **parameter** `a` in the type definition

- If the pairs have different types, we get an error
  e.g., `let list3 = LoP[(1,'a'),(2,'b'),(3,'c')]` yields an error
Polymorphism in User-Defined Types/2

- If you need the pairs to store different kinds of types, you have to use different type variables

```haskell
data AdvListOfPairs a b = ALoP [(a,b)] deriving (Show)

> let list1 = ALoP[(1,'a'),(2,'b')]
> list1
ALoP [(1,'a'),(2,'b')]
> let list2 = ALoP[(1,2),(2,3),(3,4)]
> list2
ALoP [(1,2),(2,3),(3,4)]
```
You can have recursive types in Haskell

Let's look at an example: defining a polymorphic tree structure

```haskell
data Tree a = Nil | Node a (Tree a) (Tree a)
    deriving (Show)
```

```haskell
let tree1 = Nil
> tree1
Nil

> let tree2 = Node 'a' (Node 'b' Nil Nil) (Node 'c' Nil Nil)
> tree2
Node 'a' (Node 'b' Nil Nil) (Node 'c' Nil Nil)
```
Pattern matching can be used to access individual nodes and sub-trees

```haskell
data Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show)

> let tree = Node 'a' (Node 'b' Nil Nil) (Node 'c' Nil Nil)
> let (Node val child1 child2) = tree
> val 'a'
> child1 (Node 'b' Nil Nil)
> let (Node v c1 c2) = child1
> v 'b'
> c1 Nil
```
Operating on recursive types often needs recursive functions as well.

If we want to determine the depth of a tree, we could do it like this:

\[
\text{depth} :: \text{Tree} \ a \rightarrow \text{Int} \\
\text{depth} \ \text{Nil} = 0 \\
\text{depth} (\text{Node} \ a \ \text{left} \ \text{right}) = 1 + \text{max} \ (\text{depth} \ \text{left}) \ (\text{depth} \ \text{right})
\]

The first case is straightforward: an empty tree has depth 0.

The second case traverses the tree recursively and adds one to the depth of the deeper subtree.

A tail-recursive version of the depth function:

\[
\text{depthTR} :: \text{Tree} \ a \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{depthTR} \ \text{Nil} \ n = n \\
\text{depthTR} (\text{Node} \ a \ \text{l} \ \text{r}) \ n = \text{max} \ (\text{depthTR} \ l \ n+1) \ (\text{depthTR} \ r \ n+1)
\]
Traversal of a Tree

- **Preorder traversal**
  
  ```haskell
  preorder :: Tree a -> [a]
  preorder Nil = []
  preorder (Node a l r) = a : (preorder l) ++ (preorder r)
  ```

- **Postorder traversal**
  
  ```haskell
  postorder :: Tree a -> [a]
  postorder Nil = []
  postorder (Node a l r) = a : (postorder l) ++ (postorder r)
  ```

- **Inorder traversal**
  
  ```haskell
  inorder :: Tree a -> [a]
  inorder Nil = []
  inorder (Node a l r) = (inorder l) ++ [a] ++ (inorder r)
  ```
Outline

1. Functions
2. User-Defined Data Types
3. Type Classes Revisited
Recall that type classes define which operations can work on which inputs (similar to interfaces in other programming languages).

- That is, a type class provides function signatures.
- A type is an instance of a (type) class if it supports all functions of that class.

We are now going to have another look at type classes.

So far we’ve automatically made some of our types instances of existing type classes with the keyword deriving.

- e.g., data ListOfPairs a = LoP [(a,a)] deriving (Show)

We will now

- make a type instance of a type class explicitly, which includes also the definition of some functions (Haskell may not always be able to derive them automatically as in the case of the type class Show).
- create our own type classes.
Let’s build a simple enumerated type called TrafficLight:

```haskell
data TrafficLight = Red | Yellow | Green
```

We want this type to be comparable, i.e., be an instance of `type class Eq`, which is defined as follows:

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

- `x == y = not (x /= y)`
- `x /= y = not (x == y)`

The keyword `class` introduces a new type class and the overloaded operations, which must be supported by any type that is an instance of that class.

The last two lines mean that Haskell can figure out the definition of the other function, i.e., only one of the two need actually to be implemented.
Creating an Instance of a Type Class/2

In order to make TrafficLight an instance of Eq, we have to
  • declare TrafficLight an instance of Eq using the keyword instance
  • declare one of the two functions (==) or (/=)

```
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
  Red == Red = True
  Green == Green = True
  Yellow == Yellow = True
  _ == _ = False
```

Now variables of type TrafficLight can be compared

```
> Red == Red
True

> Red == Green
False
```
Let’s build our own user-defined type classes

In other languages, you can use lots of different values for conditionals

For example, in JavaScript, 0 and "" evaluate to false, any other integer and non-empty string to true

To introduce this behavior into Haskell, we write a YesNo type class that takes a value and returns a Boolean value

The keyword class begins the definition of a new type class

class YesNo a where
    yesNo :: a -> Bool
Next, we’ll make Int/Integer an instance of our new type class.

This allows us to evaluate integer numbers to a boolean value.

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True

instance YesNo Integer where
    yesno 0 = False
    yesno _ = True
```

```haskell
> yesno 4
True
> yesno 0
False
```
The Functor type class is a built-in type class, which is basically for things that can be mapped, i.e., the map operator can be applied
  
  e.g., lists are an instance of this type class

How is this class defined?

class Functor f where
  fmap :: (a -> b) -> f a -> f b

This definition essentially says: give me a function \( a \to b \) and a box with \( a \)'s in it and I’ll give you a box with \( b \)'s in it

\( f \) is a type constructor, i.e., a constructor that takes a type parameter/variable to create a new type

For example, a list is a type that takes a type parameter

  - A concrete value always has to be a list of some type, e.g., a list of strings, it cannot be just a generic list
So, a functor allows us to map functions over various data types.

A functor takes

- a function of type \( \text{a} \rightarrow \text{b} \)
- and some data type \( \text{f a} \) (i.e., a type constructor \( \text{f} \) with type parameter \( \text{a} \))

and returns

- some data type \( \text{f b} \) (i.e., a type constructor with type parameter \( \text{b} \))

For example, for a list of type \( \text{a} \) and a function \( \text{a} \rightarrow \text{b} \)

- you get as return value a list of type \( \text{b} \)
- And that’s exactly what a map operator does on a list.
A List is an Instance of the Functor Type Class

- A list (\[\ldots\]\]) is an instance of the type class Functor

  instance Functor [] where
  fmap = map

- [] is a type constructor (actually the list constructor)

- Compare the signature of fmap and map

  > :t fmap
  fmap :: Functor f => (a -> b) -> f a -> f b

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

- Notice that the type constructor f is replaced by the list constructor []

- Using the map function

  > map (\x -> x * x) [1,2,3]
  [1,4,9]
Now we make Tree an instance of class Functor

```
instance Functor Tree where
    fmap f Nil = Nil
    fmap f (Node x left right) =
        Node (f x) (fmap f left) (fmap f right)
```

Doing a map on an empty tree is straightforward: it returns an empty tree

For any other tree, we have to recursively go down the left and right subtrees
Now we can run a map (more specifically an fmap) on our tree

```haskell
> let tree1 = Node 1 (Node 2 Nil Nil) (Node 3 Nil Nil)
> fmap (+2) tree1
Node 3 (Node 4 Nil Nil) (Node 5 Nil Nil)
> fmap (show) tree1
Node "1" (Node "2" Nil Nil) (Node "3" Nil Nil)
```