



# *Generic programming in Haskell*

Bo Simonsen

bo@geekworld.dk

Department of Computing  
The University of Copenhagen

June 6, 2008



## *Introduction*

What is haskell?

Short introduction

## *Generic programming*

Introduction

Classes

Inheritance

An example: A binary tree

## *References*



## *What is haskell?*

Haskell is a functional language. Some of the strengths are:

- Lazy evaluation - makes infinite lists possible
- Type classes
- Monads
- All you know from ML: Algebraic types, Pattern matching, Polymorphism



## *A short introduction to haskell*

- Functions in haskell are defined using mathematical syntax e.g.

Math:

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

$$f(x) = x^2$$

Haskell:

```
f :: Int -> Int
```

```
f x = x * x
```

- Selection is done with guards (similar to the fork-notation in math):

```
negate :: Bool -> Bool
```

```
negate x
```

```
  | x == True = False
```

```
  | otherwise = True
```



## *A short introduction to haskell II*

- Recursion is similar to ML:

```
fib :: Int -> Int
```

```
fib 0 = 1
```

```
fib 1 = 1
```

```
fib x = fib (x-1) + fib (x-2)
```



## *Generic programming*

- Generic programming is an easy task. Simply replace the type with a lowercase letter e.g. ( $\{a, \dots, z\}$  is often used)

```
rev :: [a] -> [a] -> [a]
```

```
rev [] y = y
```

```
rev (x:xs) y = rev xs (x:y)
```

This example works well but how about our first function?

- We need to use:

```
square :: Num a => a -> a
```

```
square x = x * x
```

- But why the "Num a" part? Because the (\*) operator/function requires it! But now the function works with floats, integers, ...



## *Generic Programming II*

- A error occur if I try 'square "abc"'

```
Main> square "abc"
```

```
ERROR - Cannot infer instance
```

```
*** Instance    : Num [Char]
```

```
*** Expression : square "abc"
```

## *Classes in haskell*

Classes are a mixture of concepts and operator overloading in C++.

- The num class:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)      :: a -> a -> a
  negate            :: a -> a
  abs, signum       :: a -> a
  fromInteger       :: Integer -> a
```

...

```
instance Num Int where
```

...

- The class is defining properties and the instances is implementing them (default implementations are made in the class).



## *Classes II*

- Other built-in classes:
  - Ord (ordering, less than, greater than, ..)
  - Eq (equality `==` / `≠`)
  - Real
  - Integral (modulo operations)
  - Fractional (p/q)
  - Show (to string)
  - Floating (math functions like sin, cos, tan)
  - Functor
  - Enum (sequential ordering)
  - Bounded



## *Inheritance*

- We will design a class, where we require the following for the input type
  - Ordering Ord
  - Equality Eq
  - Numeric Num
- We will then program some functions which will be possible to use within this class scope.
- The code on the next slide, is based on inheritance (Num and Ord inherits from Eq) and implement the described ideas.



## *Inheritance II*

```
class (Num a, Ord a) => Myclass a where
  ds :: a -> a
```

```
  ds x
    | x < 5 = x + 5
    | otherwise = x
```

```
instance Myclass Int
```

```
genInt :: Int
genInt = 5
```

```
doSomething :: Myclass a => a -> a
doSomething x = ds x
```



## *An example: A binary tree*

With algebraic data types we can easily construct a data structure for a binary tree:

```
data Tree a = NilT | Node a (Tree a) (Tree a)
tmp = (Node 5 (Node 3 NilT NilT) (Node 7 NilT NilT))
```

Here I define the structure and make a simple tree containing 3,5,7. But how about printing it out? Without doing anything the interpreter gives us:

```
Main> tmp
ERROR - Cannot find "show" function for:
*** Expression : tmp
*** Of type    : Tree Integer
```



## *An example: A binary tree II*

As the error message said, we need to define an instance for the show class in order to print out the tree (This is equivalent to operator overloading in C++):

```
instance Show a => Show (Tree a) where
  show (Node x l r) = show x ++ " " ++ show l ++ show r
  show (NilT) = ""
```

Now I can simply write:

```
Main> tmp
5 3 7
```



## *Compilers/Interpreters & References*

### Compilers / Interpreters

- GHC - Glasgow Haskell Compiler  
*Compiler and Interpreter*  
<http://www.haskell.org/ghc/>
- Hugs  
*Interpreter*  
<http://www.haskell.org/hugs/>

### References

- Simon Thompson - *Haskell: The Craft of Functional Programming* - 2nd edition, ISBN: 0-201-34275-8, Pearson/Addinson-Wesley.
- Richard Bird - *Introduction to Functional Programming using Haskell* - 2nd edition, ISBN: 0-13-484346-0, Pearson/Prentice Hall.