# AC21007: Haskell Lecture 6
# Tail Recursion, Algebraic Data Types, Type Classes

František Farka

# Recapitulation

- Sorting algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort

## Tail recursion

- A recursive function is tail recursive iff the final result of the recursive call is the final result of the function itself
- I.e. the outermost function applied in an RHS expression.
- Non-tail recursive `sum`:

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs)  = x + (sum xs)
```

- A tail-recursive version - we use an additional accumulator acc:

```
sumAux :: [Int] -> Int -> Int
sumAux []     acc = acc {- + 0 -}
sumAux (x:xs) acc = sumAux xs (acc + x)

sum :: [Int] -> Int
sum xs = sumAux xs 0
```

# Tail recursion - Fibonacci numbers

- Fibonacci numbers: $F_n = \begin{cases} = 0 & n = 0 \\ = 1 & n = 1 \\ = F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

  $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

- Haskell implementation is straightforward:

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

- Can we turn this into a tail-recursive function?

# Tail recursion - Fibonacci numbers (cont.)

- Observation: recursive step performs two recursive calls
- ...in `sum` it performs one r.c. and uses one `acc` ...
- ...we are going to use two intermediate values!
- An implementation:

```
fibHelper :: Int -> Int -> Int -> Int
fibHelper 0 val1    val2    = val1
fibHelper 1 val1    val2    = val2
fibHelper n val1    val2    =
    fibHelper (n - 1) val2 (val1 + val2)

fib :: Int -> Int
fib n = fibHelper n 0 1
```

# Tail recursion and folds

- We already saw folds - "schemes" of recursive functions
- We know that e.g. `sum` can be expressed as a fold:

```
sum :: [Int] -> Int
sum xs = foldl (+) 0 xs
```

  or

```
sum' :: [Int] -> Int
sum' = foldr (+) 0
```

- Is either of these tail-recursive?

# Tail recursion and folds (cont.)

- recall recursive steps of `foldr` and `foldl`:

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

...

foldl f z (x:xs) = let z' = z `f` x
                   in foldl f z' xs
```

- `foldr` is *not* tail-recursive but `foldl` is tail recursive!

# Algebraic Data Types

- We define our own data types by stating data type name and it's *constructors* – both identifiers types and constructors begin with an uppercase letter
- We already know `Bool`:

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

  `Bool` is a type with two constructors: `True` and `False`.
- Similarly we can define e.g.:

```
data Suits  = Spades
            | Hearts
            | Diamonds
            | Clubs
```

# Algebraic Data Types (cont.)

- We also saw tuples and a constructor `(,)` – e.g.:

```haskell
(1, 'c') :: (Int, Char)
```

- Constructors may contain *fields* of certain type, e.g.:

```haskell
data MyPair = MyPair Int Char
```

  Note that the name of a type and a name of it's constructor can be the same. A value of type `MyPair`:

```haskell
myPairVal :: MyPair
myPairVal = MyPair 1 'c'
```

- Values of algebraic data types are constructed in the same way as values of lists and tuples.

# Algebraic Data Types (cont.)

- We can also pattern-match on data type values in function definitions and let-bindings in the very same way as with lists and tuples:

```
incMyPair :: MyPair -> MyPair
incMyPair (MyPair i c) = MyPair (i + 1) c
```

- . . . but tuple type is more general – `(a, b)` for any types `a` and `b`

# Algebraic Data Types (cont.)

- Data types may be polymorphic in fields of constructors,

```
data Pair a b = Pair a b
```

  we can specify type variables after the name of type and use them as types of constructor fields.

- And we can combine all of the above:

```
data CrazyType a b c
    = NoParamCtor
    | MonoMorphicCtor1 Int
    | MonoCtor2 String [Char] Bool
    | MonoCtor3 MyPair
    | PolyCtor1 a b
    | PolyCtor2 a Int
    | PolyCtor3 (Pair c Int)
    | PolyCtor4 (c -> a, Int)
```

- We call these data types *Algebraic Data Types* (ADT's)

# Some old ADT's . . .

- The list type is just an ordinary type, the only special thing is syntactic sugar for "[]" and "(:)":

```haskell
data List a = Nil | Cons a (List a)

length' :: List a -> Int
length' Nil          = 0
length' (Cons _ xs) = 1 + length' xs
```

- And the same for tuples, as we already saw:

```haskell
data Pair a b = Pair a b

fst' :: Pair a b -> a
fst' (Pair x _) = x

snd' :: Pair a b -> b
snd' (Pair _ y) = y
```

# ...and some new

- Sometimes, we need an extra value:

```haskell
data Maybe a = Nothing | Just a

safeHead :: [a] -> Maybe a
safeHead []     = Nothing
safeHead (x:_)  = Just x
```

- ADT representing binary trees (values are only in leafs):

```haskell
data BinTree a
    = Leaf a
    | Node (BinTree a) (BinTree a)

myTree :: BinTree Char
myTree = Node (Leaf 'a') (Leaf 'b')
```

# Type Classes

- So far we saw monomorphic functions, e.g. `neg`, `and`, and polymorphic functions, e.g. `fst`, `head`.
- What if we want a function, that is polymorphic only for some types (ad-hoc polymorphism), e.g. `sort` for `Int`, `Integer`, and `Float`?

```
sort ::  Ord a => [a] -> [a]
sort = ...  {- (<=) for a-values -}
```

  We need to constrain type variable *a* to types, that can be ordered.
- `Ord a` is a type class *constraint*.

# Type Classes (cont.)

- We can define a *class* of types and specify which functions (called class *methods*) are available for types of this class (i.e. type class behaves as an interface), e.g.:

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

- We can specify, that a type is and *instance* of a class – we provide an implementation of class functions for this type:

```haskell
instance Ord Int where
    x <= y = primitiveIntComparison x y

instance Ord Float where
    x <= y = primitiveFloatComparison x y
```

# Type Classes (cont.)

- Instance definitions can itself be constrained and do recursively compose. Recall our ADT `List`:

```
instance Ord a => Ord (List a) where
    Nil <= _        = True
    (List x xs) <= (List y ys) =
        if (x <= y)
            then if (x == y)
                then xs <= ys
                else True
            else False
```

- And there is a similar instance for `[a]`
- That means that if we provide instance `Ord OurData` we get `Ord [OurData]` for free.

## Type Classes (cont.)

- Some standard Haskell type classes:
  - `Eq a` – types with equality, `(==)`
  - `Ord a` – ordered types, `(<), (<=)`
  - `Show a` – types that can be pretty printed using `show`
  - `Num a` – numeric types - `(+), (-), (*), abs, signum`
  - And many more ...
- Now we can fully understand type of e.g. `(+)`:

```
GHCi, version 7.10.3:
Prelude> :t (+)

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

## Strong Static Typing

- So far, we always provided a type of top-level definitions (functions)
- ...we did not provide a type of functions in e.g. where blocks
- Compiler *infers* the **most generic** type of any expression automatically and in fact we do not need to provide even types of top level definitions, e.g. for

  ```
  -- mySum :: ???
  mySum = foldr (+) 0 xs
  ```

  compiler infers the following type:

  ```
  GHCi, version 7.10.3:
  Prelude> :t mySum
  mySum :: (Num b, Foldable t) => t b -> b
  ```

- **Best Practice:** Do provide top-level types – types document functions, and help compiler produce simpler error messages

# Next time

- Monday the the 22th of February, 2-3PM, Dalhousie 3G05 LT2
- More sorting algorithms
  - Quick Sort
  - Merge Sort
- IO in Haskell (Monads)