Outline of Lecture 12

- Lazy evaluation in Haskell
- List comprehension revisited
- Infinite lists
- Algebras as Haskell type classes
- The `Monoid` and `Functor` type classes

Lazy evaluation in Haskell

- The underlying (lazy) evaluation strategy: Haskell will only evaluate an argument to a function if that argument's value is needed to compute the overall result
- If an argument is structured (e.g., a list or a tuple), only those parts of the argument that are needed for computation will be evaluated
- Since an intermediate result (e.g., list) will be only generated on demand, using such a list will not necessarily will be expensive computationally
- One of the consequences: a possibility to describe infinite data structures. Under lazy evaluation, often only parts of such a data structure need to be examined

Lazy evaluation in Haskell (cont.)

- When the Haskell evaluation process starts, a *thunk* is created for each expression
- A *thunk* – a placeholder in the underlying graph of the program. It will be evaluated (reduced), if necessary. Otherwise, the garbage collector will eventually sweep it away
- If it is evaluated, because it's in the graph, it can be shared between expressions without re-calculation
- Lazy evaluation is often compared to non-strictness

Strict vs non-strict languages

- Strict languages evaluate *inside out*; nonstrict languages like Haskell evaluate *outside in*
- *Outside in* means that evaluation proceeds from the outermost parts of expressions and works inward based on what values are needed. Thus, the order of evaluation and what gets evaluated can vary depending on inputs
- While in strict languages, evaluation starts with subexpressions. When all of them are evaluated, their enclosing expressions are calculated, etc. Thus, it goes *inside out*
- The following would work only in a nonstrict language:

```
Prelude> fst (1,undefined)
1
Prelude> tail [undefined,2,3]
[2,3]
```

Lazy evaluation and function application

- Now, let's consider different evaluation scenarios in Haskell
- The argument which is not needed for producing the overall result will not be evaluated, e.g.

```
switch ::  Integer -> a -> a -> a
switch n x y
  | n>0 = x
  | otherwise = y
```

If the integer `n` is positive, only `x` is evaluated while the value `y` is "ignored". And vice versa in the `otherwise` case

Lazy evaluation and function application

- The duplicated argument is never evaluated more than once, e.g.

```
hh ::  Integer -> Integer -> Integer
hh x y
  | x>0 = x+x
  | otherwise = ...
```

If the first guard succeeds, the value of `x` is evaluated only once (and stored in the internal Haskell data graph). For instance, in the function application

```
hh 12 (34^4 - hh 99 5)
```

the second argument expression is never evaluated

Lazy evaluation and function application

- An argument is not necessarily evaluated fully. Only the parts that are needed are examined, e.g.

```
pm ::  (Integer,Integer) -> Integer
pm (x,y) = x+1
```

If we apply this function to the pair `(3+2,4-17)`, only the first part of the pair will be fully evaluated

Evaluation order for a function application

A reminder: general form of a function declaration:

```
f p₁ p₂ … pₖ
  | g₁        = e₁
  | g₂        = e₂
  …
  | otherwise = eᵣ
   where
    l₁ a₁,₁ … = r₁
    l₂ a₂,₁ … = r₂
    …

f q₁ q₂ … qₖ
  = ...
```

where $p_i, q_i, a_{i,j}$ are argument patterns, g_i are boolean expressions, and l_i are local identifiers.

Evaluation order for a function application (cont.)

- A function declaration may contain a number of equations (with pattern matching), then a number of guarded declarations with each equation, as well as several local definitions for each equation
- While applying a function, pattern matching expressions in function equations are evaluated in the order they come (until the first success)
- Moreover, for each applied pattern, only the necessary parts of argument expressions are evaluated
- Similarly, the guards are evaluated in the defined order (until the first success)
- Only those local definitions that are needed (either in guards or result expressions) are evaluated

General evaluation order in an Haskell expression

- Evaluation is **from outside in**. In situations like

  $$\underline{\texttt{f}_1\ \texttt{e}_1\ \underline{(\texttt{f}_2\ \texttt{e}_2\ 17)}}$$

  where one application encloses another, the outer one is evaluated first

- Otherwise, evaluation is **from left to right**. In the expression like

  $$\underline{\texttt{f}_1\ \texttt{e}_1} + \underline{(\texttt{f}_2\ \texttt{e}_2)}$$

  the underlined expressions are both to be evaluated, however, the left one will be evaluated first. In some cases like `False && p`, the evaluation of the left expression is sufficient for the overall result

List comprehensions revisited

- From the evaluation order standpoint ...
- A reminder: a list comprehension is an expression of the form

$$[e \mid q_1, q_2, \ldots, q_k]$$

  where each q_i is either
  - a **generator** of the form `p <- lExp`, where `p` is a pattern and `lExp` is an expression of the list type
  - a **test**, `bExp`, which is a boolean expression
- Multiple generators allow to combine elements from two or more lists. What is the evaluation order?

List comprehensions revisited (cont.)

- Example:

```
pairs ::  [a] -> [b] -> [(a,b)]
pairs xs ys = [(x,y) | x <-xs, y<-ys]
```

  Then calling `pairs [1,2,3] [4,5]` gives us

  `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

- First, the first value from `xs`, 1, is fixed and all possible values from `ys` are chosen. Then, the process is repeated for the remaining values from `xs` (2 and 3)

List comprehensions revisited (cont.)

- This order is not accidental, since we can have the second generator to depend on the value given by the first generator, e.g:

```
triangle ::  Int -> [(Int,Int)]
triangle n = [(x,y) | x <-[1..n], y<-[1..x]]
```

Then calling `triangle 3` gives us

```
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

Thus, the value of `x` restricts how many values are considered for `y`

List comprehensions revisited (cont.)

- Example: Pythagorean triples (where the sum of squares of the first two numbers is equal to square of the third one):

```
pyTriples ::  Integer -> [(Integer,Integer,Integer)]
pyTriples n = [(x,y,z) | x <-[2..n], y<-[x+1..n],
  z <- [y+1..n], x*x + y*y == z*z]
```

Here the test combines the values from the three generators

List comprehensions revisited (cont.)

- Generators may rely on (recursive) function calls. Example of calculating permutations:

```
perms ::  Eq a => [a] -> [[a]]
perms [] = [[]]
perms xs = [x:ps | x<-xs, ps <- perms (xs\\[x])]
```

where \\ is the list subtraction (difference) operator from Data.List

List comprehensions revisited (cont.)

- If some generator patterns are **refutable**, i.e., may sometimes fail, the corresponding elements are filtered out from (not counted in) the result. For instance,

```
heads ::  [[a]] -> [a]
heads zs = [x | (x:_) <- zs]
```

  If we apply

```
> heads [[],[2],[4,5],[]]
```

  the result is simply `[2,4]`

Infinite lists

- One important consequence of lazy evaluation is a possibility for the language to describe **infinite** structures, where only the necessary finite portion will be actually evaluated
- Any recursive type will contain infinite objects. We will concentrate on infinite lists here
- A simple example:

```
ones ::  [Integer]
ones = 1 : ones
```

- Evaluation of `ones` in Haskell produces a list of ones, indefinitely:

```
[1,1,1,1,1,1,1,^C,1,1,1,Interrupted
```

Infinite lists (cont.)

- We can sensibly evaluate functions applied to `ones`, e.g.,

```
> take 20 ones
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

- Built in the system are the lists of the form `[n ..]` and `[n,m ..]` so that

```
[3 ..]   == [3,4,5,6, ...
[3,5, ..]  == [3,5,7,9, ...
```

- We can define these functions ourselves, e.g.,

```
from ::  Integer -> [Integer]
from n = n : from (n+1)
```

Infinite lists (cont.)

- List comprehensions can also define infinite lists. Example (all Pythagorean triples):

```
pyTriples = [(x,y,z) | z <- [2..], y <- [2..z-1],
  x <- [2 ..  y-1], x*x + y*y == z*z]
```

- Another example: generating prime numbers (*Sieve of Eratosthenes*):

```
primes ::  [Integer]
primes = sieve [2 ..]

sieve (x:xs) =
  x :  sieve [y | y <- xs, y `mod` x > 0]
```

  - Sieve the infinite list and then add the first "survived" element to the prime list
  - Then use this last found prime as the number to sieve on
  - Repeat indefinitely

Infinite lists (cont.)

- Example: generating pseudo-random numbers:

```
nextRand ::  Integer -> Integer
nextRand n = (multiplier*n + increment) `mod` modulus

randomSequence ::  Integer -> [Integer]
randomSequence = iterate nextRand

seed = 17489
multiplier = 25173
increment = 13849
modulus = 65536
```

```
> randomSequence seed
[17489,59134,9327,52468,43805,8378,18395, ...
```

Why infinite lists?

- **Data-directed computing** (a sequence of data generating processes and generic data transformations)
- Constructing and manipulating potentially **infinite/unlimited resources**. We don't know how much of the resource will be needed while constructing the program
- More abstract and simpler to write

Abstract patterns and algebras

- Haskell allows to recognise abstract patterns in code, which have well-defined and analysed representations in mathematics
- A word frequently used to describe these abstractions is algebra, by which we mean one or more operations and the set they operate over
- Examples of such algebras: monoids, semigroups, functors, monads, ..
- In Haskell, these algebras can be implemented with type classes
- Type classes define the set of operations, while their instances define how each operation will perform for a given type or set

Type class `Monoid`

- In mathematics, a monoid is an algebraic structure with a single associative binary operation and an identity element
- In other words, it is a data type for which we can define a binary function such as:
  - the function takes two parameters of the same type;
  - there exists such a value that does not change other values when used with the function (identity element);
  - If we have three or more values and use the function to reduce them to a single result, the application order does not matter (associativity).
- Examples: `Integer` with `(*)` and `1`, `List a` (`[a]`) with `(++)` and `[]`

Type class `Monoid` (cont.)

- The class definition:

```
class Monoid a where
  mempty ::  a
  mappend ::  a -> a -> a
  mconcat ::  [a] -> a
  mconcat = foldr mappend mempty
```

`mempty` – the identity element,
`mappend` – the binary monoid operation,
`mconcat` – generalisation of `mappend` over a list of values

Monoids are ideal for folding

Monoid examples

- Lists are monoids:

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

- Maybe a is a monoid:

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Reusing algebras

- The last example of monoid instance demonstrates that algebras can be reused:

```
instance Monoid a => Monoid (Maybe a) ...
```

- More such examples:
  - `instance Monoid b => Monoid (a -> b) ...`
  - `instance (Monoid a, Monoid b) => Monoid (a, b) ...`
  - `instance (Monoid a, Monoid b, Monoid c) => Monoid (a, b, c) ...`

Monoid laws

- Three mathematical properties (laws) that are expected from any monoid instance
- Left identity:
  `mappend mempty x = x`
- Right identity:
  `mappend x mempty = x`
- Associativity:
  `mappend x (mappend y z) = mappend (mappend x y) z`
- Validating/checking the laws for an instance candidate: with QuickCheck, ...

Functors

- Functor – pattern of mapping over or around some structure that we do not want to alter
- That is, we want to apply the function to the value that is "inside" of some structure and leave the structure intact
- Example: a function gets applied for each element of a list and the list structure remains. No elements are removed or added, only transformed
- The type class `Functor` generalises this pattern for many types of structure

Intuition behind functors

- Applying data transformations within the given context / structure / "box" / "wrapper"
- Functors encode
  - going inside the structure (list, tree, any data constructor),
  - applying the given transformation on the extracted inside values,
  - reconstructing the original structure
- Often a sequence of actions when the values are extracted from the context, transformed, and then the context is restored are needed

Haskell type class `Functor`

- The `Functor` type class: the types that can be mapped over
- The definition:

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

- The type class contains a single operation `fmap` for working within a given structure
- Looks very similar to the familiar `map`:

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

- What's a structure here? What is `f` stands for? The answers soon

fmap examples

- Looks like a whole lot of fmap is going around:

```
Prelude> fmap (*10) [2,7]
[20,70]
Prelude> fmap (+1) (Just 1)
(Just 2)
Prelude> fmap (+1) Nothing
Nothing
Prelude> fmap (+10/) (4,5)
(4,2.0)
Prelude> fmap (++ "Esq.") (Right "Chris Allen")
(Right "Chris Allen, Esq.")
```

- The same principle: transformations that happen within some external structure (a list, a tuple or a data type)

What's `f` stands for in the `fmap` type?

- There are two kinds of constructors in Haskell: type constructors and data constructors. Type constructors are used only at the type level, in type signatures and typeclass declarations and instances
- Type constructors: functions that take types and produce types. Examples: `[]`, `(,)`, `Maybe`, `Either`, `Tree` `...` User-defined data type names are also type constructors, if the type definition contains at least one type variable
- The `Functor` type class is parameterised over such a type constructor (`f`)
- Essentially, `f` introduces the structure that `fmap` works inside on!

Functor examples: Lists as Functors

- It is not coincidence that the definition of `fmap` function looks like the `map` function on lists

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

- Lists are an instance of the `Functor` type class:

```
instance Functor [] where
  fmap = map
```

- Having `[a]` instead of `[]` here would generate an error: a function on types (a type constructor) is expected, not a concrete type like `[a]`

Functor examples (cont.)

- Trees are functors too
- A version of the `map` function for trees was defined as:

```
mapTree ::  (a-> b) -> Tree a -> Tree b
mapTree Nil = Nil
mapTree f (Node x t1 t2) =
  Node (f x) (mapTree f t1) (mapTree f t2)
```

- `Tree` is a functor:

```
instance Functor Tree where
  fmap = mapTree
```

Functor examples

- Mapping through elements of some type is often required and useful feature
- Example: transmitting the error through `mapMaybe`

```
mapMaybe ::  (a->b) -> Maybe a -> Maybe b

mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)
```

- Maybe is a functor:

```
instance Functor Maybe where
  fmap = mapMaybe
```

Again, writing `Maybe`, not `Maybe a`