# Lecture 7. Case studies

## Functional Programming 2018/19

Alejandro Serrano

# Goals

Practice our Haskell skills

1. Propositions
   - Tautology checker
   - Simplification
2. Arithmetic expressions
   - Differentiation

Chapters 8.6 from Hutton's book

# Propositions

# Definition

Propositional logic is the simplest branch of logic, which studies the truth of *propositional formulae* or *propositions*

Propositions $P$ are built up from the following components:

- Basic values, $\top$ (true) and $\bot$ (false)
- Variables, $X$, $Y$, $\ldots$
- Negation, $\neg P$
- Conjunction, $P_1 \wedge P_2$
- Disjunction, $P_1 \vee P_2$
- Implication, $P_1 \implies P_2$

For example, $(X \wedge Y) \implies \neg Y$

# Truth value of a proposition

Each proposition becomes either true or false given an assignment of truth values to each of its variables

Take $(X \wedge Y) \implies \neg Y$:

- $\{$ $X$ true, $Y$ false $\}$ makes the proposition true
- $\{$ $X$ true, $Y$ true $\}$ makes the proposition false

# Tautologies

A proposition is called a **tautology** if it is true for any assignment of variables

# Tautologies

A proposition is called a **tautology** if it is true for any assignment of variables

- $X \vee \neg X \vee \neg Y$


Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

# Problem: Test for Tautologies

- Problem: Compute if a proposition is a tautology.

# Problem: Test for Tautologies

- Problem: Compute if a proposition is a tautology.
- Approach:
  1. Design a data type `Prop` to represent Propositions
  2. Write a function `tv :: Assignment -> Prop -> Bool` computes the truth value of a proposition
  3. Collect all possible assignments
  4. Write a function `taut :: Prop -> Bool` which computes if a given proposition is a tautology

# Step 1: Propositions as a data type

We can represent propositions in Haskell

```
data Prop = Basic Bool
          | Var   Char
          | Not   Prop
          | Prop :/\: Prop
          | Prop :\/: Prop
          | Prop :=>: Prop
          deriving Show
```

The example $(X \wedge Y) \implies \neg Y$ becomes

```
(Var 'X' :/\: Var 'Y') :=>: (Not (Var 'Y'))
```

# Step 1: Assignments as a data type

- How to represent assignments?

# Step 1: Assignments as a data type

- How to represent assignments?

```haskell
type Assigment = Map Char Bool
```

# Step 2: Cooking `tv`

1. Define the type

```
tv :: Assignment -> Prop -> Bool
```

2. Enumerate the cases

```
tv _ (Basic b)     = _
tv m (Var v)       = _
tv m (Not p)       = _
tv m (p1 :/\: p2)  = _
tv m (p1 :\/: p2)  = _
tv m (p1 :=>: p2)  = _
```

# Step 2: Cooking `tv`

3. Define the simple (base) cases
   - The truth value of a basic value is itself
   - For a variable, we look up its value in the map

```
tv _ (Basic b) = b
tv m (Var v)   =
    case lookup v m of
      Nothing -> error "Variable unknown!"
      Just b  -> b
```

# Step 2: Cooking `tv`

4. Define the other (recursive) cases
   - We call the function recursively and apply the corresponding Boolean operator

```
tv m (Not p)        = not (tv m p)
tv m (p1 :/\: p2) = tv m p1 && tv m p2
tv m (p1 :\/: p2) = tv m p1 || tv m p2
tv m (p1 :=>: p2) = not (tv m p1) || tv m p2
```

# Step 3: Obtaining Assignments

- Find all assignments

# Step 3: Obtaining Assignments

- Find all assignments

```
assigns :: Prop -> [Assignment]
```

# Step 3: Obtaining Assignments

- Find all assignments

```
assigns :: Prop -> [Assignment]
```

- Main idea:
  a. Obtain all the variables in the formula

  ```
  vars :: Prop -> [Char]
  ```

  b. Generate all possible assignments

  ```
  assigns' :: [Char] -> [Assignment]
  ```

# Step 3: Obtaining Assignments

- Find all assignments

```
assigns :: Prop -> [Assignment]
```

- Main idea:
  a. Obtain all the variables in the formula

  ```
  vars :: Prop -> [Char]
  ```

  b. Generate all possible assignments

  ```
  assigns' :: [Char] -> [Assignment]
  ```

```
assigns = assigns' . vars
```

# Step 3a: Cooking `vars`

1. Define the type
2. Enumerate the cases
3. Define the simple (base) cases
   - A basic value has no variables, a `Var` its own
4. Define the other (recursive) cases

```
vars :: Prop -> [Char]
vars (Basic b)     = []
vars (Var v)       = [v]
vars (Not p)       = vars p
vars (p1 :/\: p2) = vars p1 ++ vars p2
vars (p1 :\/: p2) = vars p1 ++ vars p2
vars (p1 :=>: p2) = vars p1 ++ vars p2
```

# Step 3a: Cooking `vars`

```
> vars ((Var 'X' :/\: Var 'Y') :=>: (Not (Var 'Y')))
"XYY"
```

This is not what we want, each variable should appear once

- Remove duplicates using `nub` from the Prelude

```
vars :: Prop -> [Char]
vars = nub . vars'
  where vars' (Basic b) = []
        vars' (Var v)   = [v]
        vars' ...   -- as before
```

# Step 3b: Cooking `assigns'`

1. Define the type

   ```
   assigns' :: [Char] -> [Assignment]
   ```

2. Enumerate the cases

   ```
   assigns' []     = _
   assigns' (v:vs) = _
   ```

3. Define the simple (base) cases
   - Be careful! You have *one* assignment for *zero* variables

   ```
   assigns' []     = [empty]
   ```

   - What happens if we return `[]` instead?

# Step 3b: Cooking `assigns'`

4. Define the other (recursive) cases
   - We duplicate the assignment for the rest of variables, once with the head assigned true and one with the head assigned false

```
assigns' (v:vs)
    =  [ insert v True as  | as <- assigns' vs]
    ++ [ insert v False as | as <- assigns' vs]
```

# Step 4: Checking for Tautologies

- We want a function `taut :: Prop -> Bool` which checks that a given proposition is a tautology

# Step 4: Checking for Tautologies

- We want a function `taut :: Prop -> Bool` which checks that a given proposition is a tautology
- Given the ingredients, `taut` is simple to cook

```
-- Using and :: [Bool] -> Bool
taut p = and [tv as p | as <- assigns p]
-- Using all :: (a -> Bool) -> [a] -> Bool
taut p = all (\as -> tv as p) (assigns p)
-- Using all :: (a -> Bool) -> [a] -> Bool
--   and flip :: (a -> b -> c) -> (b -> a -> c)
taut p = all (flip tv p) (assigns p)
```

# Simplification

A classic result in propositional logic

*Any proposition can be transformed to an equivalent one which uses only the operators* $\neg$ *and* $\wedge$

1. De Morgan law: $A \vee B \equiv \neg(\neg A \wedge \neg B)$
2. Double negation: $\neg(\neg A) \equiv A$
3. Implication truth: $A \implies B \equiv \neg A \vee B$

# Cooking `simp`

1. Define the type

   ```
   simp :: Prop -> Prop
   ```

2. Enumerate the cases
3. Define the simple (base) cases

   ```
   simp b@(Basic _)  = b
   simp v@(Var _)    = v
   ```

# Cooking `simp`

4. Define the other (recursive) cases
   - For negation, we simplify if we detect a double one

```
simp (Not p)      = case simp p of
                        Not q -> q
                        q     -> Not q
```

   - For conjunction we rewrite recursively

```
simp (p1 :/\: p2) = simp p1 :/\: simp p2
```

   - For disjunction and implication, we simplify an equivalent form with less operators

```
simp (p1 :\/: p2) = simp (Not (Not p1 :/\: Not p2)
simp (p1 :=>: p2) = simp (Not p1 :\/: p2)
```

# Arithmetic expressions

# Expressions as a data type

We define a Haskell data type for arithmetic expressions

```
data ArithOp   = Plus | Minus | Times | Div
                 deriving Show

data ArithExpr = Constant Integer
               | Variable Char
               | Op ArithOp ArithExpr ArithExpr
                 deriving Show
```

In contrast with propositions, we separate the name of the operations from the structure of the expression

# Evaluation

- Returns an integer value given values for the variables
- Similar to the truth value of a proposition

```haskell
eval :: Map Char Integer -> ArithExpr -> Integer
eval _ (Constant c) = c
eval m (Variable v) = case lookup v m of
    Nothing -> error "unknown variable!"
    Just x  -> x
eval m (Op o x y)   = evalOp o (eval m x) (eval m y)
  where evalOp Plus  = (+)
        evalOp Minus = (-)
        evalOp Times = (*)
        evalOp Div   = div
```

- Note that the result of `evalOp` is a function

# Differentiation

# Derivative / Afgeleide

The *derivative* of a function is another function which measures the amount of change in the output with respect to the amount of change in the input

For example, velocity is the derivative of distance with respect to time

We write $v = \dfrac{dx}{dt}$ following Leibniz's notation

# Rules for differentiation

*Differentiation* is the process of finding the derivative

We just need to follow some simple rules

$$\frac{dx}{dx} = 1 \qquad \frac{dc}{dx} = 0 \text{ if } c \text{ is constant} \qquad \frac{dy}{dx} = 0 \text{ if } y \not\equiv x$$

$$\frac{d(f \pm g)}{dx} = \frac{df}{dx} \pm \frac{dg}{dx} \qquad \frac{d(f \cdot g)}{dx} = \cdot \frac{df}{dx} \cdot g + f \cdot \frac{dg}{dx}$$

$$\frac{d\frac{f}{g}}{dx} = \frac{\frac{df}{dx} \cdot g - f \cdot \frac{dg}{dx}}{g \cdot g}$$

# Differentiation in Haskell

$$\frac{dx}{dx} = 1 \qquad \frac{dc}{dx} = 0 \text{ if } c \text{ is constant} \qquad \frac{dy}{dx} = 0 \text{ if } y \not\equiv x$$

```haskell
diff (Constant _) _ = Constant 0
diff (Variable v) x
  | v == x          = Constant 1
  | otherwise       = Constant 0
```

$$\frac{d(f \pm g)}{dx} = \frac{df}{dx} \pm \frac{dg}{dx}$$

```haskell
diff (Op Plus  f g) x
  = Op Plus  (diff f x) (diff g x)
diff (Op Minus f g) x
  = Op Minus (diff f x) (diff g x)
```

# Differentiation in Haskell

$$\frac{d(f \cdot g)}{dx} = \cdot \frac{df}{dx} \cdot g + f \cdot \frac{dg}{dx} \qquad \frac{d\frac{f}{g}}{dx} = \frac{\frac{df}{dx} \cdot g - f \cdot \frac{dg}{dx}}{g \cdot g}$$

```
diff (Op Times f g) x
  = Op Plus (Op Times (diff f x) g)
            (Op Times f (diff g x))
diff (Op Div   f g) x
  = Op Div (Op Plus (Op Times (diff f x) g)
                    (Op Times f (diff g x)))
           (Op Times g g)
```

# Symbolic manipulation

- `eval`, `simp` and `diff` manipulate expressions
  - As opposed to values such as numbers or Booleans
  - This is called *symbolic manipulation*
- Data types and pattern matching are essential to write these functions concisely
  - Functions operate as rules to rewrite expressions
- Source code can be represented in a similar way
  - The corresponding data type is big
  - For that reason, Haskell is regarded as one of the best languages to write a compiler