Lecture 7. Case studies

Functional Programming 2017/18

Alejandro Serrano

Universiteit Utrecht

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

Goals

Practice our Haskell skills

1. Propositions
   - Tautology checker
   - Simplification
2. Arithmetic expressions
   - Countdown problem
   - Differentiation
3. Reverse Polish Notation calculator

Chapters 8.6-8.7 and 9 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, ...
- Negation, $\neg P$
- Conjunction, $P_1 \wedge P_2$
- Disjunction, $P_1 \vee P_2$
- Implication, $P_1 \implies P_2$

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

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'))
```

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

Cooking `tv`

1. Define the type

```
tv :: Map Char Bool -> 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) = _
```

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)   = fromJust (lookup v m)
```

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
```

Tautologies

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

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

1. Obtain all the variables in the formula

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

2. Generate all possible assignments

   ```
   assigns :: [Char] -> [Map Char Bool]
   ```

3. Check that all the assignments leads to true

Cooking `taut`

Given the ingredients, `taut` is simple to cook

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

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
```

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
```

Cooking `assigns`

1. Define the type

```
assigns :: [Char] -> [Map Char Bool]
```

2. Enumerate the cases

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

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

```
assigns []     = [[]]
```

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

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

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
   - Use `@` patterns to prevent recomputation

```
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

```
eval :: Map Char Integer -> ArithExpr -> Integer
eval _ (Constant c) = c
eval m (Variable v) = fromJust (lookup v m)
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

Des chiffres et des lettres (1965 - present)

A popular French TV show, also known as:

- Cijfers en letters (1975 - 1988, 1989 - 1993)
- Paloriamo (1977 - 1989)
- Countdown (1982 - present)
- Cifras y letras (1991 - 1996, 2002 - 2013)
- …

Given a sequence of (usually six) numbers and a target, attempt to construct an expression whose value is the target by using values from the sequence, simple arithmetic operations and parentheses

Valid arithmetic expressions

The rules of the game restrict all operations to take and return natural numbers

- The first operand of substraction must be larger or equal than the second
- Only exact divisions are allowed

Let's refine the evaluator to allow only valid expressions

Valid arithmetic expressions

- Evaluation may fail, so we wrap the result in `Maybe`

```
eval' :: Map Char Integer -> ArithExpr
      -> Maybe Integer
```

- Constants and variables are essentially the same

```
eval' _ (Constant c) = Just c
eval' m (Variable v) = lookup v m
```

Valid arithmetic expressions

- Addition and multiplication have no side rules
  - How do we make `(+)` and `(*)` work with `Maybe Integer` instead of `Integer`?
  - Solution: *lifting* an operation along `Maybe`

```
liftM2 :: (a -> b -> c)
       -> Maybe a -> Maybe b -> Maybe c

eval' m (Op Plus  x y)
  = liftM2 (+) (eval' m x) (eval' m y)
eval' m (Op Times x y)
  = liftM2 (*) (eval' m x) (eval' m y)
```

Valid arithmetic expressions

- For substraction and division we apply the side rules
  - We check only when both subexpressions are valid

```
eval' m (Op Minus x y)
  = case (eval' m x, eval' m y) of
      (Just x, Just y) | x >= y
                       -> Just (x - y)
      _                -> Nothing
eval' m (Op Div   x y)
  = case (eval' m x, eval' m y) of
      (Just x, Just y) | y == 0
                       -> Nothing
                       | x `mod` y == 0
                       -> Just (x `div` y)
      _                -> Nothing
```

Universiteit Utrecht

A glimpse of monads

Using the fact that Maybe is a monad, we can write

```
...
eval' m (Op Plus  x y)
  = (+) <$> eval' m x <*> eval' m y
eval' m (Op Minus x y)
  = do a <- eval' m x
       b <- eval' m y
       guard (a >= b)
       return (a - b)
...
```

This style of programming is covered later in the course

Dividing the problem

The problem of finding a solution with a set of given numbers can be divided as follows

- Obtain all sequences of numbers from the given ones, in any order

  ```
  choices :: [Integer] -> [[Integer]]
  ```

- Build expressions for each of those sequences

  ```
  exprs :: [Integer] -> [Expr]
  ```

- Filter out the invalid ones and those which do not lead to the desired number

Cooking `sols`

Given those ingredients, we can cook `sols` with comprehension spices

```
sols :: [Integer] -> Integer -> [Expr]
sols given target
  = [expr | choice <- choices given
          , expr   <- exprs choice
          , eval' [] expr == Just target]
```

This is a brute force solution, not very scalable

Cooking `choices`

To build `choices` we make use of two library functions

```
subsequences :: [a] -> [[a]]
--   subsequences [1,2,3]
-- = [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

permutations :: [a] -> [[a]]
--   permutations [1,2,3]
-- = [[1,2,3],[2,1,3],[3,2,1],
--     [2,3,1],[3,1,2],[1,3,2]]
```

Cooking `choices`

- Composition is not the right answer

```
permutations . subsequences :: [a] -> [[[a]]]
```

  - We permute the list of subsequences, not each subsequence on its own

Universiteit Utrecht

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

Cooking `choices`

- Composition is not the right answer

```
permutations . subsequences :: [a] -> [[[a]]]
```

  - We permute the list of subsequences, not each subsequence on its own

- We apply under each subsequence using `map`

```
> (map permutations . subsequences) [1,2]
[[[]],[[1]],[[2]],[[1,2],[2,1]]]
```

  - We still have too many list layers, we should flatten

Cooking `choices`

- Composition is not the right answer

```
permutations . subsequences :: [a] -> [[[a]]]
```

  - We permute the list of subsequences, not each subsequence on its own

- We apply under each subsequence using `map`

```
> (map permutations . subsequences) [1,2]
[[[]],[[1]],[[2]],[[1,2],[2,1]]]
```

  - We still have too many list layers, we should flatten

- We flatten a list of lists using `concat`

```
choices = concat
        . map permutations
        . subsequences
```

Cooking `exprs`

1. Define the type

```
exprs :: [Integer] -> [ArithExpr]
```

2. Enumerate the cases

```
exprs []  = _
exprs [n] = _
exprs ns  = _
```

3. Define the simple (base) cases

   - With no number there is no expression
   - With just one number, there is only one expression

```
exprs []  = []
exprs [n] = [Constant n]
```

Cooking `exprs`

4. Define the other (recursive) cases
   - Given a sequence of numbers, we divide it in all possible pairs of non-empty lists using `splits`

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

   - Each subsequence is turned into all possible expressions
   - We combine all expressions using all operations

```
exprs ns
  = [Op op x y | (left, right) <- splits ns
               , x <- exprs left
               , y <- exprs right
               , op <- [Plus .. Div]]
```

Cooking `splits`

- Empty and one-element lists cannot be split
- If we have more than one element, `(x:xs)`
  - We can attach the head to the result of splitting the rest
  - We have the additional split `([x], xs)`

```
splits :: [a] -> [([a], [a])]
splits []  = []
splits [_] = []
splits (x:xs)
  = ([x], xs) : [(x:ls, rs) | (ls, rs) <- splits xs]
```

It works!

```
> sols [1,2,3] 6
[ Op Times (Constant 2) (Constant 3)
, Op Times (Constant 3) (Constant 2)
, Op Plus  (Constant 1)
           (Op Plus (Constant 2) (Constant 3))
, Op Times (Constant 1)
           (Op Times (Constant 2) (Constant 3))
, Op Plus  (Op Plus (Constant 1) (Constant 2))
           (Constant 3), ... ]
```

- Lots of repetition, 3 * 2 and 2 * 3
- Useless operations, like 1 * (2 * 3)

The Countdown Problem by Hutton presents solutions

Reverse Polish Notation calculator

Reverse Polish Notation (RPN)

Notation in which an operator follows its operands

```
    3 4 + 2 * 10 -
=       7 2 * 10 -
=          14 10 -
=                4
```

Parentheses are not needed when using RPN

Historical note: RPN was invented in the 1920s by the Polish mathematician Łukasiewicz, and rediscovered by several computer scientists in the 1960s

RPN expressions

Expressions in RPN are lists of numbers and operations

```
data Instr = Number Float | Operation ArithOp
type RPN   = [Instr]
```

We reuse the `ArithOp` type from arithmetic expressions

For example, `3 4 + 2 *` becomes

```
[ Number 3, Number 4, Operation Plus
, Number 2, Operation Times ]
```

RPN calculator

To compute the value of an expression in RPN, you keep a stack of values

- Each number is added at the top of the stack
- Operations use the top-most elements in the stack

```
             3           4           +           2           *

                       | 4 |                   | 2 |
        -> | 3 | -> | 3 | -> | 7 | -> | 7 | -> | 14 |
+--+     +----+    +----+    +----+    +----+    +-----+
```

Cooking `evalRPN`

We consume elements in the list left-to-right

- In other words, we *fold left*
- At the end, we recover what is at the top of the stack

```
evalRPN :: RPN -> Float
evalRPN = head . foldl evalInstr []
```

What is the type of `evalInstr`?

```
foldl :: (b -> a -> b) -> [a] -> [b]
-- the elements are of type a = Instr
-- the result is the stack  b = Stack
foldl :: (Stack -> Instr -> Stack)
      -> [Instr] -> Stack
```

Cooking `evalInstr`

```
evalInstr :: Stack -> Instr -> Stack
```

- We model a stack by a simple list, where the head is the top element

```
type Stack = [Float]
```

- We push on the stack when we find a number

```
evalInstr stack (Number f) = f : stack
```

- In the case of an operation, we pop from the stack and push the result on top

```
evalInstr (x:y:stack) (Operation op)
  = evalOp op x y : stack
    where evalOp ... -- as previously
```

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

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

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

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

```
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