A general introduction to Functional Programming using Haskell

Matteo Rossi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
rossi@elet.polimi.it
Functional programming in a nutshell

• In mathematics, a function \( f : D \rightarrow R \) takes an argument \( d \in D \) and returns a value \( r \in R \)
  – that's all it does, it does not “affect” anything else
  – it has no side effects
• The absence of side effects is the key idea behind functional programming

• In fact, one could program in a “functional” way also in classic imperative languages such as C, but functional programming languages enforce it
  – at least, “purely functional” PLs enforce it; most FPLs allow mixing functional style and imperative style
    • in some cases side effects can simplify programming considerably

• In these introductory lectures we present functional programming using Haskell as reference language
  – one of the “pure” FPLs available
• Introductory reference text: 
  \emph{Programming in Haskell}
  G. Hutton
  Cambridge University Press, 2007
First examples in Haskell

- Key mechanism in FPL: function definition
  \[
  \text{double } x = x + x
  \]

- Evaluation through function application:
  \[
  \text{double 3} = \{ \text{applying double} \}
  3 + 3 = \{ \text{applying } + \}
  6
  \]

- Absence of side effects means that evaluation order does not affect the result

- Compare
  \[
  \text{double (double 2)} = \{ \text{applying the inner double} \}
  \text{double (2 + 2)} = \{ \text{applying } + \}
  \text{double 4} = \{ \text{applying double} \}
  4 + 4 = \{ \text{applying } + \}
  8
  \]
  with
  \[
  \text{double (double 2)} = \{ \text{applying the outer double} \}
  \text{double 2 + double 2} = \{ \text{applying the first double} \}
  (2 + 2) + \text{double 2} = \{ \text{applying the first } + \}
  4 + \text{double 2} = \{ \text{applying double} \}
  4 + (2 + 2) = \{ \text{applying the second } + \}
  4 + 4 = \{ \text{applying } + \}
  8
  \]
First examples (2)

• A typical fragment of imperative programming:
  ```
  count := 0
  total := 0
  repeat
    count := count + 1
    total := total + count
  until
    count = n
  ```

• In purely functional programming there is no notion of “variable”, whose value changes as the execution progresses nor, in fact, of “loop”

• The basic mechanism for repetition in FP is recursion

• Compare with the definition of function `sum_up_to` given by the following equation:
  ```
  sum_up_to n = if n == 0 then 0 else n + sum_up_to (n-1)
  ```

• An alternative definition:
  ```
  sum_up_to2 n = sum_seq [1..n]
  where
    sum_seq [] = 0
    sum_seq (x:xs) = x + sum_seq xs
  ```
  – `sum_up_to2` is a function that takes a value `n` as input
  – it is computed by applying function `sum_seq` to the sequence of values from 1 to `n`
  – `sum_seq` is defined through the two equations that follow the `where` keyword; it is a function that is applied to sequences of values
    • if the sequence to which `sum_seq` is applied is empty, the result is 0
    • otherwise, the result of `sum_seq` is given by adding the first element of the sequence to the sum of the other elements
Quicksort in Haskell

- \texttt{qsort \{empty\} = \{empty\}}
- \texttt{qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger}
  \hspace{1cm} \textbf{where}
  \hspace{1cm} \texttt{smaller = [a \mid a \leftarrow xs, a \leq x ]}
  \hspace{1cm} \texttt{larger = [b \mid b \leftarrow xs, b > x ]}

- The two equations define that quicksort is a function that is applied to sequences of values and that:
  - if \texttt{qsort} is applied to an empty sequence, the sequence is already sorted
  - otherwise, let us call \texttt{x} the first element of the sequence, and \texttt{xs} the rest of the sequence; then, if \texttt{smaller} is the subsequence of \texttt{xs} that contains all elements no bigger than \texttt{x}, and \texttt{larger} is the subsequence of \texttt{xs} that contains all elements bigger than \texttt{x}, the sorted sequence is given by concatenating \texttt{smaller}, \texttt{x} and \texttt{larger}

- Example of execution:
  \texttt{qsort [3, 5, 1, 4, 2]}
  \hspace{1cm} \{ applying \texttt{qsort} \}
  \texttt{qsort [1, 2] ++ [3] ++ qsort [5, 4]}
  \hspace{1cm} \{ applying \texttt{qsort} \}
  \texttt{(qsort [] ++ [1] ++ qsort [2]) ++ [3]} \hspace{1cm} \{ applying \texttt{qsort, since qsort [x] = [x]} \}
  \hspace{1cm} \{ applying ++ \}
  \texttt{([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [ ])} \hspace{1cm} \{ applying ++ \}
  \texttt{[1, 2] ++ [3] ++ [4, 5]} \hspace{1cm} \{ applying ++ \}
  \texttt{[1, 2, 3, 4, 5]}
Function application: Haskell syntax

- In mathematics, if $f$ is a function that takes two arguments and $a, b, c, d$ are values, we write:
  \[ f(a, b) + cd \]
- In Haskell the same is written as:
  \[ f\ a\ b + c*d \]
  - function application has higher priority than other operators, hence the line above is equivalent to
  \[ (f\ a\ b) + c*d \]
Types in Haskell

- Haskell is statically typed
- we write \( v :: T \) to state that value \( v \) has type \( T \)
  - for example
    \[
    \text{False} :: \text{Bool} \\
    \text{not} :: \text{Bool} \to \text{Bool} \\
    \text{not False} :: \text{Bool}
    \]
  - \text{not} is a function that takes a \text{Bool} as argument and returns a \text{Bool}
- Some basic types (mostly self-explanatory):
  - \text{Bool}
  - \text{Char}
  - \text{String}
  - \text{Int}
  - \text{Integer}
    - \text{Integer} represents integer numbers with \textit{arbitrary precision}
  - \text{Float}
Lists

• Lists: sequences of elements of the same type, e.g.:
  [False, True, False] :: [Bool]
  ['a', 'b', 'c', 'd'] :: [Char]
  ['One', 'Two', 'Three'] :: [String]

• The empty list: []

• Lists of lists, e.g.:
  [['a', 'b'], ['c', 'd', 'e']] :: [[Char]]

• Operations on lists (i.e., basic functions that take lists as arguments):
  – length xs
    • number of elements in list xs
  – head xs
    • first element of list xs
  – tail xs
    • all elements of list xs except first
  – xs!!n
    • n-th element of list xs (starting from 0)
  – take n xs
    • list made of first n elements of list xs
  – drop n xs
    • list obtained by removing first n elements of list xs
  – xs ++ ys
    • list obtained by appending list ys after list xs
  – reverse xs
    • list obtained by reversing the elements of list xs
Tuples

- Tuple: finite sequence of elements of possibly different type, e.g.:
  
  (False, True) :: (Bool, Bool)
  (False, 'a', True) :: (Bool, Char, Bool)
  "Yes", True, 'a') :: (String, Bool, Char)

- empty tuple: ()

- More examples:
  
  ('a',(False,'b')) :: (Char,(Bool,Char))
  ([ 'a','b'],[False, True]) :: ([Char],[Bool])
  [('a',False), ('b',True)] :: [(Char,Bool)]
Function types

- Function: mapping from values of a certain type to values of another type, e.g.:
  
  \[
  \text{not} \quad :: \quad \text{Bool} \rightarrow \text{Bool} \\
  \text{isDigit} \quad :: \quad \text{Char} \rightarrow \text{Bool}
  \]

- Using tuples and lists no more than one argument is needed:
  
  \[
  \text{add} \quad :: \quad (\text{Int}, \text{Int}) \rightarrow \text{Int} \\
  \text{add} \ (x, \ y) = x + y
  \]

  \[
  \text{zeroto} \quad :: \quad \text{Int} \rightarrow [\text{Int}] \\
  \text{zeroto} \ n = [0..n]
  \]

- Another way of dealing with multiple arguments: functions that return functions, e.g.:
  
  \[
  \text{add'} \quad :: \quad \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\
  \text{add'} \ x \ y = x + y
  \]

  \[
  - \text{add'} \ is \ a \ function \ that \ takes \ an \ \text{Int} \ as \ argument, \ and \ returns \ a \\
  \quad \ function \ that, \ given \ another \ \text{Int}, \ returns \ an \ \text{Int}
  \]

- It works also for more than two arguments:
  
  \[
  \text{mult} \quad :: \quad \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \\
  \text{mult} \ x \ y \ z = x \ast y \ast z
  \]

- Functions such as \text{add'} and \text{mult} that take arguments one at a time are called \textit{curried}
Curried functions and partial application

• Curried functions lend themselves to partial application
  – this occurs when not all arguments are supplied to the function
    application
  – the result of partially applying arguments to a curried function is
    another function, e.g.:
    add' 1 :: Int -> Int
    • we could also define
      inc :: Int -> Int
      inc = add' 1
      then, the result of inc 10 is 11

• Note that the function arrow \(\rightarrow\) associates to the right, i.e.
  Int \(\rightarrow\) Int \(\rightarrow\) Int \(\rightarrow\) Int
  means
  Int \(\rightarrow\) (Int \(\rightarrow\) (Int \(\rightarrow\) Int))
  • Function application, instead, associates to the left, i.e.
    mult x y z
    means
    ((mult x) y) z
Polymorphic types

- length is a function that can be applied to lists of different types of elements:
  length [1, 3, 5, 7]
  length ["Yes", "No"]
  length [ isDigit , isLower , isUpper ]
- In fact, the type of length is polymorphic (and length is a polymorphic function) as it contains a type variable:
  length :: [a] -> Int
    - a is the type variable
      - type variables must start with a lowercase letter
- Other examples of polymorphic functions:
  fst :: (a, b) -> a
  head :: [a] -> a
  take :: Int -> [a] -> [a]
  zip :: [a] -> [b] -> [(a, b)]
  id :: a -> a
Type classes, overloaded types, methods

- Type class = a collection of types
  - for example, class Num contains any numeric types (e.g., Int, Float)
- If a is a type variable and C is a type class, C a is a class constraint
  - it states that type a must be an instance of type class C
- Overloaded type: a type that includes a class constraint, e.g.,
  3 :: Num a => a
  - 3 is a constant that is defined for any numeric type a
- Also, + is an overloaded function:
  (+) :: Num a => a -> a -> a
  - (+) is a (curried) function that can be applied to any pairs of values that belong to an instance of type class Num
- Other examples of overloaded functions:
  (-) :: Num a => a -> a -> a
  (\*) :: Num a => a -> a -> a
  negate :: Num a => a -> a
  abs :: Num a => a -> a
  signum :: Num a => a -> a

- In general, a type class defines methods, i.e., overloaded functions that can be applied to values of instances of the type class
- For example, type class Eq contains types that can be compared for equality and inequality; as such, it defines the following 2 methods:
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  - Bool, Char, String, Int, Integer, Float are all instances of Eq; so are list and tuple types, if their element types are instances of Eq
Class declaration

• A new class is declared using the class keyword:
  class Eq a where
  (==), (=) :: a -> a -> Bool
  x /= y = not (x == y)
  – this declaration contains a default definition for method /=, so an
  instance of Eq must only define method ==

• Example of instance of class Eq:
  instance Eq Bool where
  False == False = True
  True == True = True
  _==_ = False

• Classes can be extended to form new classes:
  class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max :: a -> a -> a
  min x y | x <= y = x
  | otherwise = y
  max x y | x <= y = y
  | otherwise = x

• To define an instance of Ord, we need to provide the
  definition of methods <, <=, > and >=:
  instance Ord Bool where
  False < True = True
  _ < _ = False
  b <= c = (b < c) || (b == c)
  b > c = c < b
  b >= c = c <= b
Other basic classes

- **Ord** – ordered types
  - it contains instances of class `Eq`, whose values in addition are totally (linearly) ordered
  - it provides the following six methods:
    
    \[
    \begin{align*}
    (<) & \;::\; a \rightarrow a \rightarrow \text{Bool} \\
    (\leq) & \;::\; a \rightarrow a \rightarrow \text{Bool} \\
    (>) & \;::\; a \rightarrow a \rightarrow \text{Bool} \\
    (\geq) & \;::\; a \rightarrow a \rightarrow \text{Bool} \\
    \text{min} & \;::\; a \rightarrow a \rightarrow a \\
    \text{max} & \;::\; a \rightarrow a \rightarrow a
    \end{align*}
    \]
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` are instances of `Ord`; so are list and tuple types, if their elements...

- **Show** – showable types
  - types whose values can be converted into strings of characters using the following method
  
  \[
  \text{show} \;::\; a \rightarrow \text{String}
  \]
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` are instances of `Show`; so are list and tuple types, if their elements...

- **Read** – readable types
  - types whose values can be obtained from strings of characters using the following method
  
  \[
  \text{read} \;::\; \text{String} \rightarrow a
  \]
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` are instances of `Read`; so are list and tuple types, if their elements...
Other basic classes (cont.)

- **Num** – numeric types
  - types that are instances of both `Eq` and `Show`, whose values are also numeric, hence methods (+), (−), (∗), `negate`, `abs`, `signum` can be applied to them
    - examples of instances: `Int`, `Integer`, `Float`

- **Integral** – integral types
  - Num plus the following methods:
    - `div :: a -> a -> a`
    - `mod :: a -> a -> a`
    - examples of instances: `Int`, `Integer`

- **Fractional** – fractional types
  - Num, plus the following methods:
    - `(/) :: a -> a -> a`
    - `recip :: a -> a`
    - examples of instances: `Float`
Mechanisms to define functions

• From previous functions:
  
  \[
  \text{even} :: \text{Integral} \ a \Rightarrow a \rightarrow \text{Bool} \\
  \text{even} \ n = n \mod\ 2 == 0 \\
  \]

  \[
  \text{splitAt} :: \text{Int} \rightarrow [a] \rightarrow ([a], [a]) \\
  \text{splitAt} \ n \ zs = (\text{take} \ n \ zs, \text{drop} \ n \ zs) \\
  \]

• Using conditional expressions
  
  \[
  \text{abs} :: \text{Int} \rightarrow \text{Int} \\
  \text{abs} \ n = \text{if} \ n >= 0 \, \text{then} \ n \, \text{else} \ -n \\
  \]

  \[
  \text{signum} :: \text{Int} \rightarrow \text{Int} \\
  \text{signum} \ n = \text{if} \ n < 0 \, \text{then} \ -1 \, \text{else} \\
  \quad \text{if} \ n == 0 \, \text{then} \ 0 \, \text{else} \ 1 \\
  \quad \text{the else branch is mandatory} \\
  \]

• Using guarded equations:
  
  \[
  \text{signum} \ n \mid \begin{array}{l} 
  n < 0 \quad = -1 \\
  n == 0 \quad = 0 \\
  \text{otherwise} \quad = 1 \\
  \end{array} \\
  \quad \text{guards are evaluated in the order in which they are written} \\
  \quad \text{otherwise (which is optional) is a synonym for True} \\
  \]
Pattern matching

- Example:
  not :: Bool -> Bool
  not False = True
  not True   = False

- Example of wildcard pattern, which matches any value:
  (&&) :: Bool -> Bool -> Bool
  True && True = True
  _ && _       = False
  - patterns are matched according to the order in which they are written

- An alternative definition for &&:
  True   && b = b
  False && _ = False
  - if the first argument is True, then the value of && coincides with the value of its second argument; if the first argument if False, the value of && is False no matter the value of the second argument

- A tuple of patterns is itself a pattern, for example:
  fst :: (a, b) -> a
  fst (x, _) = x

  snd :: (a, b) -> b
  snd (_, y) = y

- Also, a list of patterns is itself a pattern:
  first_of_3_a :: [Char] -> Bool
  first_of_3_a ['a', _, _] = True
  first_of_3_a _ = False
  - first_of_3_a evaluates to True if its argument matches a list of 3 elements, in which the first one is 'a'; if the argument matches anything else, first_of_3_a evaluates to False
List patterns

• List patterns can be written using the *constructor operator* for lists (`:`)
  – any list is constructed using `:` starting from the empty list `[]`
    • for example, the list `[10, 20, 30]` is constructed as follows:
      10 : (20 : (30 : []))
      – essentially, `[10, 20, 30]` is an abbreviation for
        10 : (20 : (30 : []))
    • `:` associates to the right, so `10 : (20 : (30 : []))` is the same as `10 : 20 : 30 : []`

• Then, `x : xs` matches any list that has at least one element, `x`, and a (possibly empty) tail `xs`
  – if we are not interested in the value of the tail, we can write the pattern as `x : _`

• Examples of definitions of functions on lists:
  
  ```haskell
  first_a :: [Char] -> Bool
  first_a ('a' : _) = True
  first_a _ = False

  null :: [a] -> Bool
  null [] = True
  null (_:_ ) = False

  head :: [a] -> a
  head (x : _) = x

  tail :: [a] -> [a]
  tail (_ : xs) = xs
  ```
Lambda expressions

- A lambda expression is used to define an anonymous function
- It is made of:
  - a pattern for each argument of the function
  - a body, which defines how the result is computed from the values of the arguments
- Examples:
  \[ x \rightarrow x+x \]
  \[ (x, y) \rightarrow x+y \]
  \[ (x:xs) \rightarrow x^2 \]
  - then, if we evaluate the expression
    \[ (x:xs) \rightarrow x^2 \] \[ [4,0,10] \]
    we obtain 16 as result
- The meaning of curried function definitions can be given in terms of lambda expressions
  - for example, the meaning of definition
    \[ \text{add } x \ y = x + y \]
    is
    \[ \text{add} = \ x \rightarrow (\ y \rightarrow x + y) \]
- Lambda expressions are very useful to define functions that are used only once
  - this often occurs for functions that are used as parameters of other functions (more on this later), e.g.:
    \[ \text{odds } n = \text{map } (\ x \rightarrow x*2+1) \ [0 .. n-1] \]
    - \text{odds} is a function that takes a number \( n \) as argument, and returns the list of the first \( n \) odd numbers
      - e.g., the result of \text{odds } 3 \ is \ [1, 3, 5] 
    - \text{map} is a function that takes two arguments, a function and a list, and returns the list obtained by applying the function to the elements of the list
      - e.g., the result of \text{map } (\ x \rightarrow x*2+1)\ [0,1,2] \ is \ [0*2+1, 1*2+1, 2*2+1], i.e., \ [1,3,5] 

List comprehensions

- Comprehension: building a set from an existing one, e.g.
  \[ \{x^2 \mid x \in \{1..5\}\} \]

- A similar notation exists in Haskell to build lists from existing ones, e.g.:
  \[ [x^2 \mid x \leftarrow [1..5]] \]
  - the result is \([1, 4, 9, 16, 25]\)
  - \(x \leftarrow [1..5]\) is a generator

- there can be more than one generator (separated by commas):
  \[ [(x, y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5]] \]
  - the result is \([(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]\)

- generators can depend on one another, e.g.:
  \[ \text{concat :: } [[a]] \rightarrow [a] \]
  \[ \text{concat xss = } [x \mid xs \leftarrow xss, x \leftarrow xs] \]

- guards can be used to filter out undesired results produced by earlier generators, e.g.:
  \[ \text{factors :: } \text{Int} \rightarrow [\text{Int}] \]
  \[ \text{factors n = } [x \mid x \leftarrow [1..n], n \mod x == 0] \]

- Another example:
  \[ \text{primes_lt :: } \text{Int} \rightarrow [\text{Int}] \]
  \[ \text{primes_lt n = } [x \mid x \leftarrow [1..n],
                    (\forall y \rightarrow \text{factors y /= } [1,y]) x] \]

- The \text{zip} function produces a new list by pairing successive elements from existing lists until one (or both) are exhausted:
  \[ \text{zip } ['a', 'b', 'c'][1,2,3,4] \]
  results in \[ [('a',1),('b',2),('c',3)] \]

- Example:
  \[ \text{positions :: } \text{Eq a => a} \rightarrow [a] \rightarrow [\text{Int}] \]
  \[ \text{positions x xs = let } n = \text{length xs}-1 \text{ in }
    [i \mid (x', i) \leftarrow \text{zip xs } [0..n],
    x' == x] \]
Some examples of recursion

- Recursion is one of the key mechanisms to define functions
- The principles are the same as for recursion in imperative languages: a recursive function is defined by re-applying itself to a “smaller argument”
- Examples of recursive functions on lists:

```
product :: Num a => [a] -> a
product [] = 1
product (x:xs) = x*product xs

length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs

(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x <= y = x : y : ys
| otherwise = y : insert x ys

isort :: Ord a => [a] -> [a]
isort [] = []
isort (x : xs) = insert x (isort xs)

zip :: [a] -> [b] -> [(a,b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop (n+1) [] = []
drop (n+1) (_:xs) = drop n xs
```
Higher-order functions

• In FPLs functions can both
  – return functions
  – take functions as arguments

• Curried functions are an example of the first kind:
  \[
  \text{add} :: \text{Int} \to (\text{Int} \to \text{Int}) \\
  \text{add} = \lambda x \to (\lambda y \to x+y)
  \]
  – \text{add} is a function that takes an Int as argument, and returns a function that, given an Int, returns an Int
  • \text{add} 7 is a function that sums 7 to the Int passed as argument

• In FPLs a function can also take another function as argument:
  \[
  \text{twice} :: (a \to a) \to a \to a \\
  \text{twice} \ f \ x = f \ (f \ x)
  \]
  – function \text{twice} takes a function \( f \) and a value \( x \) as arguments, and applies \( f \) twice (first to \( x \), then to the result of \( f \ x \))
  – for example, the result of \text{twice} \ (2*) \ 4 is 16, while the result of \text{twice} \ \text{reverse} \ [2,10,1] is [2,10,1]
map and filter

• higher-order functions are often used to perform operations on lists

• A typical example (which we have already encountered):
  
  
  map :: (a -> b) -> [a] -> [b]
  map f xs = [f x | x <- xs]
  
  – e.g., the result of
    map reverse ["abc", "def", "ghi"]
    is ["cba", "fed", "ihg"]
  – the result of map (map (+1)) [[1, 2, 3], [4, 5]] is
    [[2,3,4],[5,6]]
    • map (+1) returns a function that takes a list of numbers as argument, and returns it with values incremented by 1

• another example:
  
  filter :: (a -> Bool) -> [a] -> [a]
  filter p xs = [x | x <- xs, p x]
  
  – e.g., the result of filter even [1..10] is [2,4,6,8,10]

• an example of combination of map and filter:
  
  sumsqreven :: Integral a => [a] -> a
  sumsqreven xs = sum (map (^2) (filter even xs))
  
  – it returns the sum of the squares of the even elements of list xs
foldr

- A pattern often used to define a function $f$ that applies an operator $\oplus$ to the values of a list (with $v$ some value):
  
  $f \; [] = v$
  $f \; (x:xs) = x \oplus f \; xs$

- For example:
  
  $\text{product} \; [] = 1$
  $\text{product} \; (x : xs) = x \ast \text{product} \; xs$

  and $[] = \text{True}$
  and $(x:xs) = x \&\& \text{and} \; xs$

- The (higher-order) $\text{foldr}$ function captures this pattern:
  
  $\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
  $\text{foldr} \; f \; v \; [] = v$
  $\text{foldr} \; f \; v \; (x : xs) = f \; x \; (\text{foldr} \; f \; v \; xs)$

- Defining functions $\text{product}$ and $\text{and}$ in terms of $\text{foldr}$:
  
  $\text{product} = \text{foldr} \; (\ast) \; 1$
  $\text{and} \quad = \text{foldr} \; (\&\&) \; \text{True}$

- For example, if we apply $\text{foldr} \; (\ast) \; 1$ to list
  $1:(2:(3:(4:[])))$ we obtain $1 \ast (2 \ast (3 \ast (4 \ast 1)))$

- another example of using $\text{foldr}$ to define a function:
  
  $\text{length} = \text{foldr} \; (\_ \_ \_ v \rightarrow 1+v) \; 0$

- In general, the behavior of $\text{foldr}$ is:
  
  $\text{foldr} \; (\oplus) \; v \; [x0,x1,\cdots,xn] = x0\oplus(x1\oplus(\cdots(xn\oplus v)\cdots))$

  essentially, $\text{foldr}$ corresponds to the application to the elements of
  a list of an operator that associates to the right (hence, $\text{foldr}$)
foldl

- (higher-order) function foldl is the dual of foldr: it corresponds to the application of an operator that associates to the left
- It corresponds to the pattern:

  \[ f \ v \ [ ] = v \]
  \[ f \ v \ (x:xs) = f \ (v \oplus x) \ xs \]
  - argument \( v \) works as an accumulator, which evolves by applying operator \( \oplus \) with the value of the head of the list
- This is captured by the following definition:

  \[
  \text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
  \text{foldl} \ f \ v \ [ ] = v \\
  \text{foldl} \ f \ v \ (x:xs) = \text{foldl} \ f \ (f \ v \ x) \ xs
  \]
- In general, its behavior is:

  \[
  \text{foldl} \ (\oplus) \ v \ [x0, x1, \ldots, xn] = (\cdots((v \oplus x0) \oplus x1)\cdots) \oplus xn
  \]
- Examples of definitions using foldl:

  product = foldl (*) 1
  and = foldl (&&) True
  - product and and can be defined either with foldr or with foldl since they are both associative
- Another example:

  reverse = foldl (\xs x \rightarrow x:xs) []
  - for example the result of reverse [1, 2, 3] is 3 : (2 : (1 : []))
Composition of functions

• The (higher-order) composition operator \( \cdot \) takes two functions \( f \) and \( g \) as arguments, and applies \( f \) to the result obtained by applying \( g \) to its argument
  - the type of the argument of \( g \) must be the same as the type of the result of \( f \)
• In other words:
  \[
  (\cdot) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \\
  f \cdot g = \\lambda x \rightarrow f (g x)
  \]
• The composition operator can be used to make some definitions more concise: instead of
  \[
  \text{odd } n = \text{not } (\text{even } n) \\
  \text{twice } f \ x = f (f \ x) \\
  \text{sumsqreven } xs = \text{sum } (\text{map } (^2) \ (\text{filter even } xs))
  \]
  we can define:
  \[
  \text{odd } = \text{not } \cdot \text{even} \\
  \text{twice } = f.\ f \\
  \text{sumsqreven } = \text{sum}.\text{map } (^2)\cdot\text{filter even}
  \]
  – the last definition works because composition is associative:
  \[
  f \cdot (g \cdot h) = (f \cdot g) \cdot h
  \]
• the identity function \( \text{id } = \ \lambda x \rightarrow x \) is the unit for \( \cdot \), i.e.,
  for any function \( f \) we have \( f.\text{id } = \text{id}.\ f 
  \]
• We exploit \( \text{id} \) and \( \text{foldr} \) to define a composition of lists of functions:
  \[
  \text{compose } :: [a \rightarrow a] \rightarrow a \rightarrow a \\
  \text{compose } = \ \text{foldr} \ (\cdot) \ \text{id}
  \]
Type declarations

• The simplest way to declare a type is as a synonym of an existing type:
  
  \[
  \begin{align*}
  \textbf{type} \quad \text{String} & = \text{[Char]} \\
  \textbf{type} \quad \text{Pos} & = (\text{Int}, \text{Int}) \\
  \textbf{type} \quad \text{Board} & = [\text{Pos}]
  \end{align*}
  \]

• Type parameters are admitted:
  
  \[
  \begin{align*}
  \textbf{type} \quad \text{Assoc} \ k \ v & = [(k, v)] \\
  \end{align*}
  \]
  
  - \text{Assoc} represents, through a list, a lookup table of \langle\text{key},\text{value}\rangle pairs, where the keys are of type \( k \) and the values of type \( v \)
  
  - a function that, given a key and a lookup table, returns the value associated with the key:
    
    \[
    \begin{align*}
    \texttt{find} :: \text{Eq} \ k \Rightarrow \ k \rightarrow \text{Assoc} \ k \ v \rightarrow v \\
    \texttt{find} \ k \ t & = \text{head}[v \mid (k', v) \leftarrow t, k == k']
    \end{align*}
    \]

• In Haskell one can also declare entirely new types, through a \textit{data} declaration, e.g.:

  \[
  \begin{align*}
  \textbf{data} \quad \text{Bool} & = \text{False} \mid \text{True}
  \end{align*}
  \]

• New types can be used in functions:

  \[
  \begin{align*}
  \textbf{data} \quad \text{Move} & = \text{Left} \mid \text{Right} \mid \text{Up} \mid \text{Down} \\
  \texttt{move} :: \text{Move} \rightarrow \text{Pos} \rightarrow \text{Pos} \\
  \texttt{move} \ \text{Left} \ (x,y) & = (x{-}1, y) \\
  \texttt{move} \ \text{Right} \ (x,y) & = (x{+}1, y) \\
  \texttt{move} \ \text{Up} \ (x,y) & = (x, y{+}1) \\
  \texttt{move} \ \text{Down} \ (x,y) & = (x, y{-}1)
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{moves} :: [\text{Move}] \rightarrow \text{Pos} \rightarrow \text{Pos} \\
  \text{moves} \ [] \ p & = p \\
  \text{moves} \ (m : ms) \ p & = \text{moves} \ ms \ (\text{move} \ m \ p)
  \end{align*}
  \]
Type declarations with parameters

• Constructors in `data` declarations can have parameters:
  
  ```haskell
data Shape = Circle Float | Rect Float Float
```

• Examples of functions on type `Shape`:
  
  ```haskell
square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rect x y) = x * y
  
  – notice that we can use pattern matching with the constructors
  
```

• `Circle` and `Rect` are constructor functions: their results are values of type `Shape`
  
  – `Circle 1.0` is a value onto itself, of type `Shape`, it is not evaluated any further

• `data` declarations can also have (type) parameters, e.g.:
  
  ```haskell
data Maybe a = Nothing | Just a
```

  – type `Maybe` represents optional values (i.e., values that may fail): if the value is undefined, then its value is `Nothing`, otherwise it is `Just v` (with `v` a value of type `a`)

  – example of use of type `Maybe`: “safe” functions that, in case of errors, simply return a `Nothing` value:
    
    ```haskell
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
    
    safehead :: [a] -> Maybe a
    safehead [] = Nothing
    safehead xs = Just (head xs)
    ```
Recursive types

- Types defined through `data` declarations can be recursive:
  ```haskell
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```
  - functions on type `Tree`:
    ```haskell
    occurs :: Eq a => a -> Tree a -> Bool
    occurs v (Leaf x) = v == x
    occurs v (Node t1 x t2) = v == x ||
                              occurs v t1 ||
                              occurs v t2
    
    flatten :: Tree a -> [a]
    flatten (Leaf v) = [v]
    flatten (Node t1 v t2) = flatten t1 ++ [v]
                              ++ flatten t2
    ```
Extended example: tautology checker

- (subset of) Propositional logic:
  - constants: False, True
  - propositional variables: A, B, C, ... Z
  - connectives: \(\neg\) (not), \(\land\) (and), \(\Rightarrow\) (imply)
  - parentheses: (,)
    - examples of formulas:
      \[ \neg A \land (A \land B) \Rightarrow A \land (A \Rightarrow (A \land B)) \]
    - tautology: a formula that is true, no matter the values of the propositional variable
      - for example, \((A \land B) \Rightarrow A\) is a tautology

- We declare a type for propositions:
  \[
  \text{data Prop} = \begin{array}{l}
  \text{Const Bool} \\
  \text{Var Char} \\
  \text{Not Prop} \\
  \text{And Prop Prop} \\
  \text{Imply Prop Prop}
  \end{array}
  \]

- The value of a proposition depends on the values assigned to its variables; we use a type \text{Subst} to represent possible assignments of values to variables, through a lookup table:
  \[
  \text{type Subst} = \begin{array}{l}
  \text{Assoc Char Bool}
  \end{array}
  \]
  - example of assignment: [('A', False),('B',True)]
Tautology checker (cont.)

• We define a function that, given a proposition and an assignment of values to variables, returns the value of the proposition in that assignment:
  \[
  \text{eval} :: \text{Subst} \rightarrow \text{Prop} \rightarrow \text{Bool} \\
  \text{eval } _{-} \text{ (Const } b \text{)} = b \\
  \text{eval } s \text{ (Var } v \text{)} = \text{find } v \text{ } s \\
  \text{eval } s \text{ (Not } p \text{)} = \text{not (eval } s \text{ } p \text{)} \\
  \text{eval } s \text{ (And } p_1 \text{ } p_2 \text{)} = \text{eval } s \text{ } p_1 \land \text{eval } s \text{ } p_2 \\
  \text{eval } s \text{ (Imply } p_1 \text{ } p_2 \text{)} = \text{eval } s \text{ } p_1 \leq \text{eval } s \text{ } p_2
  \]

• We define a function that returns all variables in a proposition:
  \[
  \text{vars} :: \text{Prop} \rightarrow \text{[Char]} \\
  \text{vars } (\text{Const } b) = \text{[]} \\
  \text{vars } (\text{Var } v) = \text{[} v \text{]} \\
  \text{vars } (\text{Not } p) = \text{vars } p \\
  \text{vars } (\text{And } p_1 \text{ } p_2) = \text{vars } p_1 \cup \text{vars } p_2 \\
  \text{vars } (\text{Imply } p_1 \text{ } p_2) = \text{vars } p_1 \cup \text{vars } p_2
  \]

• A function that removes duplicates from a list:
  \[
  \text{rmdups} :: \text{Eq } a \rightarrow [a] \rightarrow [a] \\
  \text{rmdups } [ ] = [ ] \\
  \text{rmdups } (x:x:s) = x : \text{rmdups } (\text{filter } (\neq x) \text{ } s)
  \]
A function that, given a number \( n \), returns all possible combinations of Booleans of length \( n \):

\[
\text{bools :: Int} \rightarrow [[\text{Bool}]]
\]

\[
\text{bools 0} = []
\]

\[
\text{bools } n+1 = \text{(map (False:) bss)++(map (True:) bss)}
\]

\[
\text{where bss = bools } n
\]

- e.g., the result of \( \text{bools 2} \) is
  
  \[
  [[\text{False,False}],[\text{False,True}],[\text{True,False}],[\text{True,True}]]
  \]

A function that generates all possible assignments to the variables of a proposition:

\[
\text{substs :: Prop} \rightarrow [\text{Subst}]
\]

\[
\text{substs } p = \text{map (zip vs) (bools (length vs))}
\]

\[
\text{where vs = rmdups (vars } p)\]

Finally, the desired function:

\[
\text{isTaut :: Prop} \rightarrow \text{Bool}
\]

\[
\text{isTaut } p = \text{and [eval } s p \mid s <- (\text{substs } p)]
\]

In fact the result of

\[
\text{isTaut (Imply (And (Var 'A') (Var 'B')) (Var 'A'))}
\]

is True