Principles of Programming Languages (H)

Matteo Pradella

November 24, 2023

Overview

1. Introduction on purity and evaluation
2. Basic Haskell
3. More advanced concepts

A bridge toward Haskell

- We will consider now some basic concepts of Haskell, by implementing them in Scheme:
- What is a *pure* functional language?
- *Non-strict* evaluation strategies
- *Currying*

What is a **functional** language?

- In mathematics, **functions** do not have **side-effects**
- e.g. if $f : \mathbb{N} \to \mathbb{N}$, $f(5)$ is a fixed value in $\mathbb{N}$, and do not depend on *time* (also called **referential transparency**)
- this is clearly not true in conventional programming languages, Scheme included
- Scheme is *mainly* functional, as programs are **expressions**, and computation is evaluation of such expressions
- but some expressions have **side-effects**, e.g. `vector-set!`
- Haskell is **pure**, so we will see later how to manage inherently side-effectful computations (e.g. those with I/O)

Evaluation of functions

- We have already seen that, in **absence of side effects** (purely functional computations) from the point of view of the result the **order** in which functions are applied **does not matter** (almost).
- However, it matters in other aspects, consider e.g. this function:

```
(define (sum-square x y)
     (+ (* x x)
        (* y y)))
```

Evaluation of functions (Scheme)

- A possible evaluation:

```scheme
(sum-square (+ 1 2) (+ 2 3))
;; applying the first +
= (sum-square 3 (+ 2 3))
;; applying +
= (sum-square 3 5)
;; applying sum-square
= (+ (* 3 3)(* 5 5))
...
= 34
```

Evaluation of functions (alio modo)

```
(sum-square (+ 1 2) (+ 2 3))
;; applying sum-square
= (+ (* (+ 1 2)(+ 1 2))(* (+ 2 3)(+ 2 3)))
...
= 34
```

- The two evaluations differ in the **order** in which function applications are evaluated.
- A function application ready to be performed is called a **reducible expression** (or **redex**)

Evaluation strategies: call-by-value

- in the first example of evaluation, redexes are evaluated according to a (leftmost) **innermost strategy**
- i.e., when there is more than one redex, the leftmost one that does not contain other redexes is evaluated
- e.g. in `(sum-square (+ 1 2) (+ 2 3))` there are 3 redexes: `(sum-square (+ 1 2) (+ 2 3)))`, `(+ 1 2)` and `(+ 2 3)` the innermost that is also leftmost is `(+ 1 2)`, which is applied, giving expression `(sum-square 3 (+ 2 3))`
- in this strategy, **arguments** of functions are always evaluated **before** evaluating the function itself – this corresponds to passing arguments **by value**.
- note that Scheme does not require that we take the *leftmost*, but this is very common in mainstream languages

Evaluation strategies: call-by-name

- a dual evaluation strategy: redexes are evaluated in an **outermost** fashion
- we start with the redex that is **not contained in any other redex**, i.e. in the example above, with `(sum-square (+ 1 2) (+ 2 3))`, which yields `(+ (* (+ 1 2)(+ 1 2))(* (+ 2 3)(+ 2 3)))`
- in the outermost strategy, functions are always **applied before their arguments**, this corresponds to passing arguments **by name** (like in Algol 60).

Termination and call-by-name

- e.g. first we define the following two simple functions:

```
(define (infinity)
  (+ 1 (infinity)))

(define (fst x y) x)
```

- consider the expression (fst 3 (infinity)):
  - Call-by-value: (fst 3 (infinity)) = (fst 3 (+ 1 (infinity))) = (fst 3 (+ 1 (+ 1 (infinity)))) = ...
  - Call-by-name: (fst 3 (infinity)) = 3

Termination and call-by-name (ii)

- If there is an evaluation for an expression that terminates, **call-by-name terminates**, and produces the same result (this result is called **Church-Rosser confluence**)
- intuitively, we can see the evaluation as working on a tree (the expression):
- with call-by-value we expand leaves, and we could be stuck in an infinite branch
- with call-by-name we expand from the root, and go level by level, so if there is a way to reach a solution, we find it.

Haskell is lazy: call-by-need

- In call-by-name, if the argument is not used, it is never evaluated; if the argument is used several times, it is **re-evaluated each time**
- **Call-by-need** is a **memoized** version of call-by-name where, if the function argument is evaluated, that value is **stored for subsequent uses**
- In a "pure" (effect-free) setting, this produces the same results as call-by-name, and it is usually faster

Call-by-need implementation: macros and thunks

- we saw that macros are different from function, as they do not evaluate and are expanded at **compile time**
- a possible idea to overcome the nontermination of `(fst 3 (infinity))`, could be to use **thunks** to prevent evaluation, and then **force** it with an explicit call
- indeed, there is already an implementation in Racket based on **delay** and **force**
- we'll see how to implement them with macros and thunks

Delay and force: call-by-name and by-need

- Delay is used to return a **promise** to execute a computation (implements **call-by-name**)
- moreover, it caches the result (**memoization**) of the computation on its first evaluation and returns that value on subsequent calls (implements **call-by-need**)

Promise

```
(struct promise
        (proc   ; thunk or value
         value? ; already evaluated?
         ) #:mutable)
```

Delay (code)

```
(define-syntax delay
  (syntax-rules ()
    ((_ (expr ...))
     (promise (lambda ()
                (expr ...)) ; a thunk
              #f)))) ; still to be evaluated
```

Force (code)

- **force** is used to force the evaluation of a promise:

```
(define (force prom)
  (cond
    ; is it already a value?
    ((not (promise? prom)) prom)
    ; is it an evaluated promise?
    ((promise-value? prom) (promise-proc prom))
    (else
      (set-promise-proc! prom
                         ((promise-proc prom)))
      (set-promise-value?! prom #t)
      (promise-proc prom))))
```

Examples

```
(define x (delay (+ 2 5))) ; a promise
(force x) ;; => 7

(define lazy-infinity (delay (infinity)))
(force (fst 3 lazy-infinity))          ; => 3
(fst 3 lazy-infinity)                  ; => 3
(force (delay (fst 3 lazy-infinity))) ; => 3
```

- here we have call-by-need only if we make every function call a promise
- in Haskell call-by-need is the default: if we need call-by-value, we need to *force* the evaluation (we'll see how)

Currying

- in Haskell, functions have only **one** argument!
- this is not a limitation, because functions with more arguments are **curried**
- we see here in Scheme what it means. Consider the function:

```
(define (sum-square x y)
     (+ (* x x)
        (* y y)))
```

- it has signature `sum-square` : $\mathbb{C}^2 \to \mathbb{C}$, if we consider the most general kind of numbers in Scheme, i.e. the complex field

Currying (cont.)

- curried version:

```
(define (sum-square x)
    (lambda (y)
      (+ (* x x)
         (* y y))))

;; shorter version:
(define ((sum-square x) y)
    (+ (* x x)
       (* y y)))
```

- it can be used *almost* as the usual version: `((sum-square 3) 5)`
- the curried version has signature `sum-square` $: \mathbb{C} \to (\mathbb{C} \to \mathbb{C})$
  i.e. $\mathbb{C} \to \mathbb{C} \to \mathbb{C}$ ($\to$ is right associative)

Currying in Haskell

- Notice how this form makes partial function application very natural.
- In Haskell every function is automatically curried and consequently managed
- the name *currying*, coined by Christopher Strachey in 1967, is a reference to logician Haskell Curry
- the alternative name *Schönfinkelisation* has been proposed as a reference to Moses Schönfinkel but didn't catch on

Haskell

- Born in 1990, designed by committee to be:
  - **purely** functional
  - **call-by-need** (sometimes called **lazy evaluation**)
  - strong **polymorphic** and **static** typing
- Standards: Haskell '98 and '10
  - *de facto* standard: the **Glasgow Haskell Compiler (GHC)**, with many extensions
- Motto: "Avoid success at all costs"
  - ex. usage: Google's *Ganeti* cluster virtual server management tool
- Beware! There are many *bad* tutorials on Haskell and monads, in particular, available online

A taste of Haskell's syntax

- more complex and "human" than Scheme: parentheses are optional!
- function call is similar, though: `f x y` stands for `f(x,y)`
- there are infix operators and are made of non-alphabetic characters (e.g. *, +, but also <++>)
- `elem` is $\in$. If you want to use it infix, just use ‘elem‘
- -- this is a comment
- lambdas: `(lambda (x y) (+ 1 x y))` is written `\x y -> 1+x+y`

Types!

- Haskell has **static** typing, i.e. the type of everything must be known at **compile time**
- there is **type inference**, so usually we do not need to explicitly declare types
- *has type* is written :: instead of : (the latter is **cons**)
- e.g.
  - `5 :: Integer`
  - `'a' :: Char`
  - `inc :: Integer -> Integer`
  - `[1, 2, 3] :: [Integer]` – equivalent to 1:(2:(3:[]))
  - `('b', 4) :: (Char, Integer)`
  - strings are **lists of characters**

Function definition

- functions are declared through a sequence of *equations*
- e.g.

```
inc n = n + 1

length :: [Integer] -> Integer
length []       = 0
length (x:xs) = 1 + length xs
```

- this is also an example of **pattern matching**
- arguments are matched with the right parts of equations, top to bottom
- if the match succeeds, the function body is called

Parametric Polymorphism

- the previous definition of `length` could work with any kind of lists, not just those made of integers
- indeed, if we omit its type declaration, it is inferred by Haskell as having type

```
length :: [a] -> Integer
```

- lower case letters are **type variables**, so [a] stands for *a list of elements of type a, for any a*

Main characteristics of Haskell's type system

- every well-typed expression is guaranteed to have a **unique principal type**
  - it is (roughly) the *least general type that contains all the instances of the expression*
  - e.g. length :: a -> Integer is too general, while length :: [Integer] -> Integer is too specific
- Haskell adopts a variant of the **Hindley-Milner** type system (used also in ML variants, e.g. F#)
- and the principal type can be **inferred automatically**
- Ref. paper: L. Cardelli, *Type Systems*, 1997

User-defined types

- are based on **data declarations**

```
-- a "sum" type (union in C)
data Bool  = False | True
```

- **Bool** is the (nullary) **type constructor**, while **False** and **True** are **data constructors** (nullary as well)
- data and type constructors live in separate name-spaces, so it is possible (and common) to use the same name for both:

```
-- a "product" type (struct in C)
data Pnt a = Pnt a a
```

- if we apply a data constructor we obtain a **value** (e.g. Pnt 2.3 5.7), while with a type constructor we obtain a **type** (e.g. Pnt Bool)

Recursive types

- classical recursive type example:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

- e.g. data constructor Branch has type:

```
Branch :: Tree a -> Tree a -> Tree a
```

- An example tree:

```
aTree = Branch (Leaf 'a')
               (Branch (Leaf 'b') (Leaf 'c'))
```

- in this case `aTree` has type `Tree Char`

Lists are recursive types

- Of course, also lists are recursive. Using Scheme jargon, they could be defined by:

```
data List a = Null | Cons a (List a)
```

- but Haskell has special syntax for them; in "pseudo-Haskell":

```
data [a] = [] | a : [a]
```

- [] is a data and type constructor, while : is an infix data constructor

An example function on Trees

```
fringe :: Tree a -> [a]

fringe (Leaf x) = [x]
fringe (Branch left right) = fringe left ++
                             fringe right
```

- (++) denotes list concatenation, what is its type?

Syntax for fields

- as we saw, *product types* (e.g. `data Point = Point Float Float`) are like **struct** in C or in Scheme (analogously, *sum types* are like **union**)
- the access is positional, for instance we may define accessors:

```
pointx Point x _ = x
pointy Point _ y = y
```

- _ stands for *don't care*
- there is a C-like syntax to have **named fields**:

```
data Point = Point {pointx, pointy :: Float}
```

- this declaration automatically defines two field names `pointx`, `pointy`
- and their corresponding **selector functions**

Type synonyms

- are defined with the keyword **type**
- some examples

```
type String = [Char]

type Assoc a b = [(a,b)]
```

- usually for readability or shortness

More on functions and currying

- Haskell has **map**, and it can be defined as:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- we can partially apply also infix operators, by using parentheses: `(+ 1)` or `(1 +)` or `(+)`

```
map (1 +) [1,2,3]  -- => [2,3,4]
```

REPL

- :t at the prompt is used for getting **type**, e.g.

```
Prelude> :t (+1)
(+1) :: Num a => a -> a
Prelude> :t +
<interactive>:1:1: parse error on input '+'
Prelude> :t (+)

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

- **Prelude** is the standard library
- we'll see later the exact meaning of **Num a =>** with **type classes**. Its meaning here is that **a** must be a **numerical type**

Function composition and $

- (.) is used for composing functions (i.e. (f.g)(x) is f(g(x)))

```
Prelude> let dd = (*2) . (1+)
Prelude> dd 6
14
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- $ syntax for avoiding parentheses, e.g. (10*) (5+3) = (10*) $ 5+3

Infinite computations

- call-by-need is very convenient for dealing with **never-ending computations** that provide data
- here are some simple example functions:

```
ones = 1 : ones

numsFrom n = n : numsFrom (n+1)

squares = map (^2) (numsFrom 0)
```

- clearly, we cannot evaluate them (why?), but there is **take** to get *finite slices* from them
- e.g.

```
take 5 squares = [0,1,4,9,16]
```

Infinite lists

- Convenient syntax for creating infinite lists:
- e.g. `ones` before can be also written as `[1,1..]`
- `numsFrom 6` is the same as `[6..]`
- **zip** is a useful function having type
  `zip :: [a] -> [b] -> [(a, b)]`

```
zip [1,2,3] "ciao"
-- => [(1,'c'),(2,'i'),(3,'a')]
```

- list comprehensions

```
[(x,y) | x <- [1,2], y <- "ciao"]
-- => [(1,'c'),(1,'i'),(1,'a'),(1,'o'),
       (2,'c'),(2,'i'),(2,'a'),(2,'o')]
```

Infinite lists (cont.)

- a list with all the Fibonacci numbers
  (note: `tail` is `cdr`, while `head` is `car`)

```
fib = 1 : 1 :
      [a+b | (a,b) <- zip fib (tail fib)]
```

A more complex example

- We are going to use the following functions:
  - any :: (a -> Bool) -> [a] -> Bool
  - takeWhile :: (a -> Bool) -> [a] -> [a]
  - e.g. `any (>0) [3,-3..(-30)]` is true;
    `takeWhile (> 0) [3,-3..(-30)]` is [3]
- Prime numbers:

```
isprime n = not . any (\x -> mod n x == 0) .
                  takeWhile (\x -> x^2 <= n) $
                  primelist
primelist = 2 : [x | x <- [3,5..], isprime x]
```

Error

- **bottom** (aka $\perp$) is defined as `bot = bot`
- all errors have value `bot`, a value shared by all types
- `error :: String -> a` is strange because is polymorphic only in the output
- the reason is that it returns **bot** (in practice, an exception is raised)

Pattern matching

- the matching process proceeds top-down, left-to-right
- patterns may have **boolean guards**

```
sign x | x > 0  = 1
       | x == 0 = 0
       | x < 0  = -1
```

- e.g. definition of **take**

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

Take and definition

- the order of definitions **matters**:

```
Prelude> :t bot
bot :: t
Prelude> take 0 bot
[]
```

- on the other hand, `take bot []` does not terminate
- what does it change, if we swap the first two defining equations?

Case

- **take** with **case**:

```
take m ys = case (m,ys) of
              (0,_)    -> []
              (_,[])   -> []
              (n,x:xs) -> x : take (n-1) xs
```

If

- **if** is available in Haskell: its syntax is
  if <**c**> **then** <**t**> **else** <**e**>.
- Using call-by-need we could also define it as a normal function:

```
if :: Bool -> a -> a -> a
if True  x _ = x
if False _ y = y
```

let and where

- **let** is like Scheme's letrec*:

```
let x = 3
    y = 12
in x+y  -- => 15
```

- **where** can be convenient to scope binding over equations, e.g.:

```
powset set = powset' set [[]] where
  powset' [] out = out
  powset' (e:set) out = powset' set (out ++
                               [ e:x | x <- out ])
```

- layout is like in Python, with meaningful whitespaces, but we can also use a C-like syntax:

```
let {x = 3 ; y = 12} in x+y
```

Call-by-need and memory usage

- **fold-left** is efficient in Scheme, because its definition is naturally **tail-recursive**:

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- *note: in Racket it is defined with (f x z)*
- this is not as efficient in Haskell, because of call-by-need:
  - foldl (+) 0 [1,2,3]
  - foldl (+) (0 + 1) [2,3]
  - foldl (+) ((0 + 1) + 2) [3]
  - foldl (+) (((0 + 1) + 2) + 3) []
  - (((0 + 1) + 2) + 3) = 6

Haskell is too lazy: an interlude on strictness

- There are various ways to enforce **strictness** in Haskell (analogously there are classical approaces to introduce laziness in strict languages)
- e.g. on data with **bang patterns** (a datum marked with ! is considered **strict**)

```
data Complex = Complex !Float !Float
```

- there are extensions for using ! also in function parameters

Forcing evaluation

- Canonical operator to **force evaluation** is seq :: a -> t -> t
- `seq x y` returns *y*, **only if** the evaluation of *x* **terminates** (i.e. it performs *x* then returns *y*)
- a strict version of **foldl** (available in *Data.List*)

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

- strict versions of standard functions are usually primed

Special syntax for **seq**

- There is a convenient *strict* variant of $ (function application) called $!
- here is its definition:

```
($!) :: (a -> b) -> a -> b
f $! x = seq x (f x)
```

Modules

- not much to be said: Haskell has a simple module system, with **import**, **export** and namespaces
- a very simple example

```
module CartProd where  --- export everything
infixr 9 -*-
-- right associative
-- precedence goes from 0 to 9, the strongest
x -*- y = [(i,j) | i <- x, j <- y]
```

Modules (cont.)

- import/export

```
module Tree ( Tree(Leaf,Branch), fringe ) where
data Tree a  = Leaf a | Branch (Tree a) (Tree a)
fringe :: Tree a -> [a] ...
```

```
module Main (main) where
import Tree ( Tree(Leaf,Branch) )
main = print (Branch (Leaf 'a') (Leaf 'b'))
```

Modules and Abstract Data Types

- modules provide the only way to build abstract data types (ADT)
- the characteristic feature of an ADT is that the representation type is hidden: all operations on the ADT are done at an abstract level which does not depend on the representation
- e.g. a suitable ADT for binary trees might include the following operations:

```
data Tree a     -- just the type name
leaf            :: a -> Tree a
branch          :: Tree a -> Tree a -> Tree a
cell            :: Tree a -> a
left, right     :: Tree a -> Tree a
isLeaf          :: Tree a -> Bool
```

ADT implementation

```
module TreeADT (Tree, leaf, branch, cell,
                 left, right, isLeaf) where
data Tree a          = Leaf a | Branch (Tree a) (Tree a)
leaf                 = Leaf
branch               = Branch
cell  (Leaf a)       = a
left  (Branch l r)   = l
right (Branch l r)   = r
isLeaf   (Leaf _)    = True
isLeaf   _           = False
```

- in the export list the type name Tree appears without its constructors
  - so the only way to build or take apart trees outside of the module is by using the various (abstract) operations
  - the advantage of this information hiding is that at a later time we could change the representation type without affecting users of the type

Type classes and overloading

- we already saw *parametric polymorphism* in Haskell (e.g. in **length**)
- **type classes** are the mechanism provided by Haskell for *ad hoc* polymorphism (aka **overloading**)
- the first, natural example is that of numbers: 6 can represent an integer, a rational, a floating point number...
- e.g.

```
Prelude> 6 :: Float
6.0
Prelude> 6 :: Integer -- unlimited
6
Prelude> 6 :: Int   -- fixed precision
6
Prelude> 6 :: Rational
6 % 1
```

Type classes: equality

- also numeric operators and equality work with different kinds of numbers
- let's start with equality: it is natural to define equality for many types (but not every one, e.g. functions - it's undecidable)
- we consider here only **value equality**, not **pointer equality** (like Java's == or Scheme's **eq?**), because pointer equality is clearly *not referentially transparent*
- let us consider **elem**

```
x `elem`  []              = False
x `elem` (y:ys)           = x==y || (x `elem` ys)
```

- its type should be: a -> [a] -> Bool. But this means that (==) :: a -> a -> Bool, even though equality is not defined for every type

class Eq

- **type classes** are used for overloading: a class is a "container" of overloaded operations
- we can declare a type to be an **instance** of a type class, meaning that it implements its operations
- e.g. class Eq

```
class Eq a where
  (==)  :: a -> a -> Bool
```

- now the type of (==) is

```
(==)   :: (Eq a) => a -> a -> Bool
```

- `Eq a` is a *constraint* on type *a*, it means that *a* must be an instance of *Eq*

Defining instances

- e.g. `elem` has type (Eq a) => a -> [a] -> Bool
- we can define instances like this:

```
instance (Eq a) => Eq (Tree a) where
-- type a must support equality as well
 Leaf x == Leaf y =  x == y
 (Branch l1 r1) == (Branch l2 r2) = (l1==l2) && (r1==r2)
  _ == _  =  False
```

- an implementation of (==) is called a **method**
- *CAVEAT* do not confuse all these concepts with the homonymous concepts in OO programming: there are similarities but also big differences

Haskell vs Java concepts

Haskell	Java
Class	Interface
Type	Class
Value	Object
Method	Method

- in Java, an Object is an *instance* of a Class
- in Haskell, a Type is an *instance* of a Class

Eq and Ord in the Prelude

- Eq offers also a standard definition of $\neq$, derived from (==):

```
class  Eq a  where
  (==), (/=) :: a -> a -> Bool
  x /= y  =  not (x == y)
```

- we can also extend Eq with comparison operations:

```
class  (Eq a) => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

- Ord is also called a **subclass** of Eq
- it is possible to have **multiple inheritance**: class (X a, Y a) => Z a

Another important class: Show

- it is used for **showing**: to have an instance we must implement **show**
- e.g., functions do not have a standard representation:

```
Prelude> (+)
<interactive>:2:1:
    No instance for (Show (a0 -> a0 -> a0))
      arising from a use of ‘print’
    Possible fix:
      add an instance declaration for (Show (a0 -> a0 -> a0))
```

- well, we can just use a trivial one:

```
instance Show (a -> b) where
  show f = "<< a function >>"
```

Showing Trees

- we can also represent binary trees:

```
instance Show a => Show (Tree a) where
  show (Leaf x) = show x
  show (Branch x y) = "<" ++ show x ++ " | " ++ show y ++ ">"
```

- e.g.

```
Branch
    (Branch
      (Leaf 'a') (Branch (Leaf 'b') (Leaf 'c')))
    (Branch
        (Leaf 'd') (Leaf 'e'))
```

- is represented as

```
<<'a' | <'b' | 'c'>> | <'d' | 'e'>>
```

Deriving

- usually it is not necessary to explicitly define instances of some classes, e.g. Eq and Show
- Haskell can be quite smart and do it automatically, by using **deriving**
- for example we may define binary trees using an infix syntax and automatic Eq, Show like this:

```
infixr 5 :^:
data Tr a =  Lf a  |  Tr a :^: Tr a
             deriving (Show, Eq)
```

- e.g.

```
*Main> let x = Lf 3 :^: Lf 5 :^: Lf 2
*Main> let y = (Lf 3 :^: Lf 5) :^: Lf 2
*Main> x == y
False
*Main> x
Lf 3 :^: (Lf 5 :^: Lf 2)
```

An example with class Ord

- Rock-paper-scissors in Haskell

```
data RPS = Rock | Paper | Scissors deriving (Show, Eq)

instance Ord RPS where
  x <= y | x == y       = True
  Rock     <= Paper     = True
  Paper    <= Scissors = True
  Scissors <= Rock      = True
  _        <= _         = False
```

- note that we only needed to define (<=) to have the instance

An example with class Num

- a simple re-implementation of rational numbers

```
data Rat = Rat !Integer !Integer deriving Eq

simplify (Rat x y) = let g = gcd x y
                     in Rat (x `div` g) (y `div` g)
makeRat x y = simplify (Rat x y)

instance Num Rat where
  (Rat x y) + (Rat x' y') = makeRat (x*y'+x'*y) (y*y')
  (Rat x y) - (Rat x' y') = makeRat (x*y'-x'*y) (y*y')
  (Rat x y) * (Rat x' y') = makeRat (x*x') (y*y')
  abs (Rat x y)           = makeRat (abs x) (abs y)
  signum (Rat x y)        = makeRat (signum x * signum y) 1
  fromInteger x           = makeRat x 1
```

An example with class Num (cont.)

- Ord:

```
instance Ord Rat where
  (Rat x y) <= (Rat x' y') = x*y' <= x'*y
```

- a better show:

```
instance Show Rat where
  show (Rat x y) = show x ++ "/" ++ show y
```

- note: Rationals are in the Prelude!
- moreover, there is class Fractional for / (not covered here)
- but we could define our version of division as follows:

```
x // (Rat x' y') = x * (Rat y' x')
```

Input/Output is dysfunctional

- what is the type of the standard function **getChar**, that gets a character from the user? `getChar :: theUser -> Char`?
- first of all, it is not **referentially transparent**: two different calls of **getChar** could return different characters
- In general, IO computation is based on **state change** (e.g. of a file), hence if we perform a **sequence of operations**, they must be performed in **order** (and this is not easy with **call-by-need**)

Input/Output is dysfunctional (cont.)

- `getChar` can be seen as a function `:: Time -> Char`.
- indeed, it is an **IO action** (in this case for Input):
  `getChar :: IO Char`
- quite naturally, to print a character we use **putChar**, that has type:
  `putChar :: Char -> IO ()`
- **IO** is an instance of the **monad** class, and in Haskell it is considered as an **indelible stain of impurity**

A very simple example of an IO program

- **main** is the default entry point of the program (like in C)

```
main = do {
  putStr "Please, tell me something>";
  thing <- getLine;
  putStrLn $ "You told me \"" ++ thing ++ "\".";
  }
```

- special syntax for working with IO: **do**, `<-`
- we will see its real semantics later, used to define an IO action as an **ordered sequence** of IO actions
- `"<-"` (note: not =) is used to obtain a value from an IO action
- types:

```
main    :: IO ()
putStr  :: String -> IO ()
getLine :: IO String
```

Command line arguments and IO with files

- compile with e.g. **ghc readfile.hs**

```
import System.IO
import System.Environment

readfile = do  {
  args <- getArgs; -- command line arguments
  handle <- openFile (head args) ReadMode;
  contents <- hGetContents handle; -- note: lazy
  putStr contents;
  hClose handle;
  }
main = readfile
```

- **readfile stuff.txt** reads "stuff.txt" and shows it on the screen
- **hGetContents** reads lazily the contents of the file

Exceptions and IO

- Of course, purely functional Haskell code can raise exceptions: *head [], 3 'div' 0*, ...
- but if we want to catch them, we need an IO action:
- `handle :: Exception e => (e -> IO a) -> IO a -> IO a`;
  the 1st argument is the *handler*
- Example: we catch the errors of **readfile**

```
import Control.Exception
import System.IO.Error
...
main = handle handler readfile
       where handler e
               | isDoesNotExistError e =
                 putStrLn "This file does not exist."
               | otherwise =
                 putStrLn "Something is wrong."
```

Other classical data structures

- What about usual, practical data structures (e.g. arrays, hash-tables)?
- Traditional versions are imperative! If really needed, there are libraries with imperative implementations living in the IO monad
- Idiomatic approach: use **immutable** arrays (`Data.Array`), and maps (`Data.Map`, implemented with balanced binary trees)
- **find** are respectively $O(1)$ and $O(\log n)$; **update** $O(n)$ for arrays, $O(\log n)$ for maps
- of course, the update operations *copy* the structure, do not change it

Example code: Maps

```
import Data.Map

exmap = let m = fromList [("nose", 11), ("emerald", 27)]
            n = insert "rug" 98 m
            o = insert "nose" 9 n
        in (m ! "emerald", n ! "rug", o ! "nose")
```

- `exmap` evaluates to `(27,98,9)`

Example code: Arrays

- `(//)` is used for update/insert
- `listArray`'s first argument is the **range** of indexing (in the following case, indexes are from 1 to 3)

```
import Data.Array

exarr = let m = listArray (1,3) ["alpha","beta","gamma"]
            n = m // [(2,"Beta")]
            o = n // [(1,"Alpha"), (3,"Gamma")]
        in (m ! 1, n ! 2, o ! 1)
```

- `exarr` evaluates to `("alpha","Beta","Alpha")`

How to reach Monads

- We saw that IO is a type constructor, instance of *Monad*
- But we still do not know what a Monad is
- Recent versions of GHC make the trip a bit longer, because we need first to introduce the following classes:
  - Foldable (not required, but useful)
  - Functor
  - Applicative (Functor)

Class Foldable

- **Foldable** is a class used for *folding*, of course
- The main idea is the one we know from *foldl* and *foldr* for lists:
- we have a container, a binary operation f, and we want to apply f to all the elements in the container, starting from a value z.
- Recall their definitions:

1. ```
   foldr f z []     = z
   foldr f z (x:xs) = f x (foldr f z xs)
   ```
2. ```
   foldl f z []     = z
   foldl f z (x:xs) = foldl f (f z x) xs
   ```

foldl vs foldr in Haskell

- A minimal implementation of Foldable requires *foldr*
- *foldl* can be expressed in term of *foldr* (*id* is the identity function):
  `foldl f a bs = foldr (\b g x -> g (f x b)) id bs a`
- to see how this works, let's try the definition on a small list [1,2]:

```
foldl f 0 [1,2] =
= foldr (\b g x -> g (f x b)) id [1,2] 0 =
(calling F the lambda and applying foldr)
= (F 1 (F 2 id)) 0 =
= (F 2 id) (f x 1) 0 =
= id (f x 2) (f x 1) 0 =
= (f x 2) (f x 1) 0 =
= (f (f x 1) 2) 0 =
= (f (f 0 1) 2)
```

foldl vs foldr in Haskell

- the converse is not true, since *foldr* may work on infinite lists, unlike *foldl*:
- in the presence of call-by-need evaluation, *foldr* will immediately return the application of f to the recursive case of folding over the rest of the list
- if f is able to produce some part of its result without reference to the recursive case, then the recursion will stop
- on the other hand, *foldl* will immediately call itself with new parameters until it reaches the end of the list.

Example: foldable binary trees

- Let's go back to our binary trees

```
data Tree a = Empty | Leaf a | Node (Tree a) (Tree a)
```

- we can easily define a *foldr* for them

```
tfoldr f z Empty = z
tfoldr f z (Leaf x) = f x z
tfoldr f z (Node l r) = tfoldr f (tfoldr f z r) l

instance Foldable Tree where
    foldr = tfoldr

> foldr (+) 0 (Node (Node (Leaf 1) (Leaf 3)) (Leaf 5))
9
```

Maybe

- **Maybe** is used to represent computations that may fail: we either have *Just v*, if we are lucky, or *Nothing*.
- It is basically a simple "conditional container"

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

- It is adopted in many recent languages, to avoid NULL and limit exceptions usage.
- Examples are Scala (basically the ML family approach): Option[T], with values None or Some(v); Swift, with Optional<T>.
- It is quite simple, so we will use it in our examples with Functors & C.

Of course, Maybe is foldable

```
instance Foldable Maybe where
    foldr _ z Nothing = z
    foldr f z (Just x) = f x z
```

Functor

- **Functor** is the class of all the types that offer a *map* operation
- (so there is an analogy with Foldable vs folds)
- the map operation of functors is called **fmap** and has type:
- `fmap :: (a -> b) -> f a -> f b`
- it is quite natural to define map for a container, e.g.:

```
instance  Functor Maybe  where
    fmap _ Nothing       = Nothing
    fmap f (Just a)      = Just (f a)
```

Functor laws

- Well-defined functors should obey the following laws:
- *fmap id = id* (where *id* is the identity function)
- *fmap (f . g) = fmap f . fmap g* (homomorphism)
- You can try, as an exercise, to check if the functors we are defining obey the laws

Trees can be functors, too

- First, let us define a suitable *map* for trees:

```
tmap f Empty = Empty
tmap f (Leaf x) = Leaf $ f x
tmap f (Node l r) = Node (tmap f l) (tmap f r)
```

- That's all we need:

```
instance Functor Tree where
    fmap = tmap

-- example
> fmap (+1) (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)
```

Applicative Functors

- In our voyage toward monads, we must consider also an extended version of functors, i.e. *Applicative functors*
- The definition looks indeed exotic:

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- note that `f` is a type constructor, and `f a` is a Functor type
- moreover, `f` must be parametric with one parameter
- if f is a container, the idea is not too complex:
  - pure takes a value and returns an f containing it
  - <*> is like fmap, but instead of taking a function, takes an f containing a function, to apply it to a suitable container of the same kind

Maybe is an Applicative Functor

- Here is its definition:

```
instance Applicative Maybe where
    pure = Just

    Just f  <*> m        = fmap f m
    Nothing <*> _        = Nothing
```

Lists

- Of course, lists are instances of Foldable and Functor. What about Applicative?
- For that, it is first useful to introduce **concat**
- `concat :: Foldable t => t [a] -> [a]`
- So we start from a container of lists, and get a list with the *concatenation* of them:
- `concat [[1,2],[3],[4,5]]` is [1,2,3,4,5]
- it can be defined as: `concat l = foldr (++) [] l`
- its composition with *map* is called **concatMap**

```
concatMap f l = concat $ map f l
> concatMap (\x -> [x, x+1]) [1,2,3]
[1,2,2,3,3,4]
```

Lists are instances of Applicative

- With concatMap, we get the standard implementation of <*> for lists:

```
instance Applicative [] where
    pure x    = [x]
    fs <*> xs = concatMap (\f -> map f xs) fs
```

- What can we do with it? For instance we can apply list of operations to lists:

```
> [(+1),(*2)] <*> [1,2,3]
[2,3,4,2,4,6]
```

- Note that we *map* the operations in sequence, then we *concatenate* the resulting lists

Trees and Applicative

- Following the list approach, we can make our binary trees an instance of Applicative Functors
- First, we need to define what we mean by tree concatenation:

```
tconc Empty t = t
tconc t Empty = t
tconc t1 t2 = Node t1 t2
```

- now, concat and concatMap (here `tconcmap` for short) are like those of lists:

```
tconcat t = tfoldr tconc Empty t
tconcmap f t = tconcat $ tmap f t
```

Applicative Trees

- Here is the natural definition (practically the same of lists):

```
instance Applicative Tree where
  pure = Leaf
  fs <*> xs = tconcmap (\f -> tmap f xs) fs
```

- Let's try it:

```
> (Node (Leaf (+1))(Leaf (*2))) <*>
   Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)

Node (Node (Node (Leaf 2) (Leaf 3))
           (Leaf 4))
     (Node (Node (Leaf 2) (Leaf 4))
           (Leaf 6))
```

A peculiar type class: Monad

- introduced by Eugenio Moggi in 1991, a monad is a kind of **algebraic data type** used to represent computations (instead of data in the domain model) - we will often call these computations **actions**
- monads allow the programmer to **chain** actions together to build an **ordered sequence**, in which each action is **decorated with additional processing rules** provided by the monad and performed automatically
- monads are **flexible** and **abstract**. This makes some of their *applications* a bit hard to understand.

A peculiar type class: Monad (cont.)

- monads can **also** be used to make **imperative** programming easier in a pure functional language
- in practice, through them it is possible to define an **imperative sub-language** on top of a purely functional one
- there are many examples of monads and tutorials (many of them quite bad) available in the Internet

The Monad Class

```
class Applicative m => Monad m where
    -- Sequentially compose two actions, passing any value produced
    -- by the first as an argument to the second.
    (>>=)       :: m a -> (a -> m b) -> m b
    -- Sequentially compose two actions, discarding any value produced
    -- by the first, like sequencing operators (such as the semicolon)
    -- in imperative languages.
    (>>)        :: m a -> m b -> m b
    m >> k = m >>= \_ -> k
    -- Inject a value into the monadic type.
    return      :: a -> m a
    return      = pure
    -- Fail with a message.
    fail        :: String -> m a
    fail s      = error s
```

The Monad Class (cont.)

- Note that only >>= is required, all the other methods have standard definitions
- >>= and >> are called **bind**
- `m a` is a *computation* (or action) resulting in a value of type a
- **return** is by default **pure**, so it is used to create a single monadic action. E.g. `return 5` is an action containing the value 5.
- **bind** operators are used to compose actions
  - `x >>= y` performs the computation `x`, takes the resulting value and passes it to `y`; then performs `y`.
  - `x >> y` is analogous, but "throws away" the value obtained by `x`

Maybe is a Monad

- Its definition is straightforward

```
instance  Monad Maybe  where
    (Just x) >>= k       = k x
    Nothing  >>= _       = Nothing
    fail _               = Nothing
```

Examples with **Maybe**

- The information managed automatically by the monad is the "bit" which encodes the **success** (i.e. *Just*) or failure (i.e. *Nothing*) of the action sequence
- e.g. `Just 4 >> Just 5 >> Nothing >> Just 6` evaluates to `Nothing`
- a variant: `Just 4 >>= Just >> Nothing >> Just 6`
- another: `Just 4 >> Just 1 >>= Just` (what is the result in this case?)

The monadic laws

- for a monad to behave correctly, method definitions must obey the following laws:
- 1) *return* is the **identity element**:

```
(return x) >>= f   <=>   f x
m >>= return       <=>   m
```

- 2) **associativity** for binds:

```
(m >>= f) >>= g    <=>     m >>= (\x -> (f x >>= g))
```

- (monads are analogous to **monoids**, with `return` $= 1$ and `>>=` $= \cdot$)

Example: monadic laws application with Maybe

- ```
  > (return 4 :: Maybe Integer) >>= \x -> Just (x+1)
  Just 5
  > Just 5 >>= return
  Just 5
  ```
- ```
  > (return 4 >>= \x -> Just (x+1))
              >>= \x -> Just (x*2)
  Just 10
  > return 4 >>= (\y ->
                    ((\x -> Just (x+1)) y)
                  >>= \x -> Just (x*2))
  Just 10
  ```

Syntactic sugar: the **do** notation

- The **do** syntax is used to avoid the explicit use of >>= and >>
- The essential translation of **do** is captured by the following two rules:

```
do e1 ; e2          <=>     e1 >> e2
do p <- e1 ; e2     <=>     e1 >>= \p -> e2
```

- note that they can also be written as:

```
do e1                        do p <- e1
   e2                           e2
```

- or:

```
do { e1 ;                    do { p <- e1 ;
     e2 }                         e2 }
```

Caveat: **return** does not return

- IO is a build-in monad in Haskell: indeed, we used the *do* notation for performing IO
- there are some catches, though – it looks like an imperative sub-language, but its semantics is based on bind and pure
- For example:

```
esp :: IO Integer
esp = do x <- return 4
         return (x+1)

> esp
5
```

The List Monad

- **List**: monadic binding involves joining together a set of calculations for each value in the list
- In practice, *bind* is `concatMap`

```
instance Monad [] where
    xs >>= f =  concatMap f xs
    fail _ = []
```

- The underlying idea is to represent *non-deterministic computations*

Lists: do vs comprehensions

- list comprehensions can be expressed in *do* notation
- e.g. this comprehension

```
[(x,y) | x <- [1,2,3], y <- [1,2,3]]
```

- is equivalent to:

```
do x <- [1,2,3]
   y <- [1,2,3]
   return (x,y)
```

the List monad (cont.)

- we can rewrite our example:

```
do x <- [1,2,3]
   y <- [1,2,3]
   return (x,y)
```

- following the monad definition:

```
[1,2,3] >>= (\x -> [1,2,3] >>=
                   (\y ->
                     return (x,y)))
```

- that is:

```
concatMap f0 [1,2,3]
where f0 x = concatMap f1 [1,2,3]
             where f1 y = [(x,y)]
```

Monadic Trees

- We can now to define our own monad with binary trees
- Knowing about lists, it is not too hard:

```
instance Monad Tree where
    xs >>= f = tconcmap f xs
    fail _   = Empty
```

Now some examples

- Monads are abstract, so monadic code is very flexible, because it can work with **any** instance of Monad
- A simple monadic comprehension:

```
exmon :: (Monad m, Num r) => m r -> m r -> m r
exmon m1 m2 = do x <- m1
                 y <- m2
                 return $ x-y
```

Let's apply it to lists and trees

- First, we try with lists:

```
> exmon [10, 11] [1, 7]
[9,3,10,4]
```

- on trees is not much different

```
> exmon (Node (Leaf 10) (Leaf 11))  (Node (Leaf 1) (Leaf 7))
Node (Node (Leaf 9) (Leaf 3))
     (Node (Leaf 10) (Leaf 4))
```

Not just simple containers

- Monads can be used to implement parsers, continuations, . . .
- and, of course, IO
- Let's try exmon with `IO Int`:

```
-- read is like in Scheme, here is used to parse the number
exmon (do putStr "?> "
          x <- getLine;
          return (read x :: Int))
      (return 10)
```

- What is the result, if we enter 12?

The State monad

1. we saw that monads are useful to automatically manage **state**
2. (e.g. think about the IO monad)
3. we now define a general monad to do it – btw it is already available in the libraries (see *Control.Monad.State*)
4. first of all, we define a type to represent our state:

   ```
   data State st a = State (st -> (st, a))
   ```
5. the idea is having a type that represent a computation with a *state*, i.e. a function taking the current state and returning the next (type *a* is the *explicit* part of the monad)
6. remember that we need *unary* type constructors! The "container" has now type constructor `State st`, because *State* has two parameters

State as a functor

1. First, we know that we need to make *State* an instance of *Functor*:

```
instance Functor (State st) where
  fmap f (State g) = State (\s -> let (s', x) = g s
                                  in  (s', f x))
```

2. the idea is quite simple: in a value of type `State st a` we apply *f* to the value of type *a* (like in all the other examples)

State as an applicative functor

1. Then, we need to make *State* an instance of *Applicative*:

```
instance Applicative (State st) where
  pure x = State (\t -> (t, x))

  (State f) <*> (State g) =
    State (\s0 -> let (s1, f') = f s0
                      (s2, x)  = g s1
                  in  (s2, f' x))
```

2. the idea is similar to the previous one: we apply *f :: State st (a -> b)* to the data part of the monad

The State monad

1. The same approach can be used for the monad definition:

```
instance Monad (State state) where
  State f >>= g = State (\olds ->
                            let (news, value) = f olds
                                State f' = g value
                            in f' news)
```

2. Reminder:

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

Running the State monad

1. An important aspect of this monad is that monadic code does not get evaluated to data, but to a function! (Note that *State* is a function and *bind* is function composition)
2. In particular, we obtain a function of the *initial state*
3. To get a value out of it, we need to call it:

```
runStateM :: State state a -> state -> (state, a)
runStateM (State f) st = f st
```

A first toy example

1. this is an old one, but it was in a different monad

```
ex = runStateM
     (do x <- return 5
         return (x+1))
     333
```

2. what is the result of evaluating *ex*?

A note on IO

1. It should be clear that, as it is, the state is not really used in a computation: it is only passed around unchanged
2. But now we could use this approach to model *time*:
3. *Int* for state, and we increment it by one at every action performed

```
data PseudoIO a = PseudoIO (Int -> (Int, a))

instance Monad PseudoIO where
  PseudoIO f >>= g = PseudoIO (\time ->
                          let (time', value) = f time
                               PseudioIO f' = g value
                          in f' (time'+1))
```

Utilities

1. To actually use the state, we need a way of accessing it
2. The point is to move the state to the data part and back, if we want to modify it in the program
3. This is easily done with these two utilities:

```
getState = State (\state -> (state, state))
putState new = State (\_ -> (new, ()))
```

Another toy example

1. let's go back and change a little bit our *ex* code:

```
ex' = runStateM
        (do x <- getState
            return (x+1))
        333
```

2. what is its evaluation?

Yet another toy example

1. another variant with *putState*:

```
ex'' = runStateM
       (do x <- getState
           putState (x+1)
           x <- getState
           return x)
       333
```

2. again, what is its evaluation?

Application: back to trees

1. We want to visit a tree and to give a number (e.g. a *unique identifier*) to each leaf
2. it is of course possible to do it directly, but we need to define functions passing the current value of the id around, to be assigned and then incremented for the next leaf
3. but we can also see this id as a *state*, and obtain we a more elegant and general definition by using our State monad

A monadic map for trees

1. first we need a *monadic map* for trees:

```
mapTreeM f (Leaf a) = do
  b <- f a
  return (Leaf b)
mapTreeM f (Branch lhs rhs) = do
  lhs' <- mapTreeM f lhs
  rhs' <- mapTreeM f rhs
  return (Branch lhs' rhs')
```

Types

1. as far as its type is concerned, we could declare it to be:

```
mapTreeM :: (a -> State state b) -> Tree a ->
            State state (Tree b)
```

2. on the other hand, if we omit the declaration, it is inferred by the compiler as:

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
```

3. this is clearly more general, and means that **mapTreeM** could work with *every monad*

Assigning numbers to leaves

1. It is now easy to do our job:

```
numberTree tree = runStateM (mapTreeM number tree) 1
    where number v = do cur <- getState
                        putState (cur+1)
                        return (v,cur)
```

Example

1. Let's try it with an example tree:

```
testTree = Branch (Branch
                   (Leaf 'a')
                   (Branch
                    (Leaf 'b')
                    (Leaf 'c')))
                  (Branch
                    (Leaf 'd')
                    (Leaf 'e'))
```

```
snd $ numberTree testTree
```

2. we obtain:

```
Branch (Branch (Leaf ('a',1))
               (Branch (Leaf ('b',2))
                       (Leaf ('c',3))))
       (Branch (Leaf ('d',4)) (Leaf ('e',5)))
```

Another application: logging

1. In this case, instead of changing the tree, we want to implement a *logger*, that, while visiting the data structure, keeps track of the found data
2. this is quite easy, if we see the *log text* as the *state* of the computation:

```
logTree tree = runStateM (mapTreeM collectLog tree) "Log\n"
    where collectLog v = do
            cur <- getState
            putState (cur ++ "Found node: " ++ [v] ++ "\n")
            return v
```

Example

1. Let's try it with our example tree:

```
putStr $ fst $ logTree testTree

Log
Found node: a
Found node: b
Found node: c
Found node: d
Found node: e
```

Legal stuff