advanced programming tutorial 3

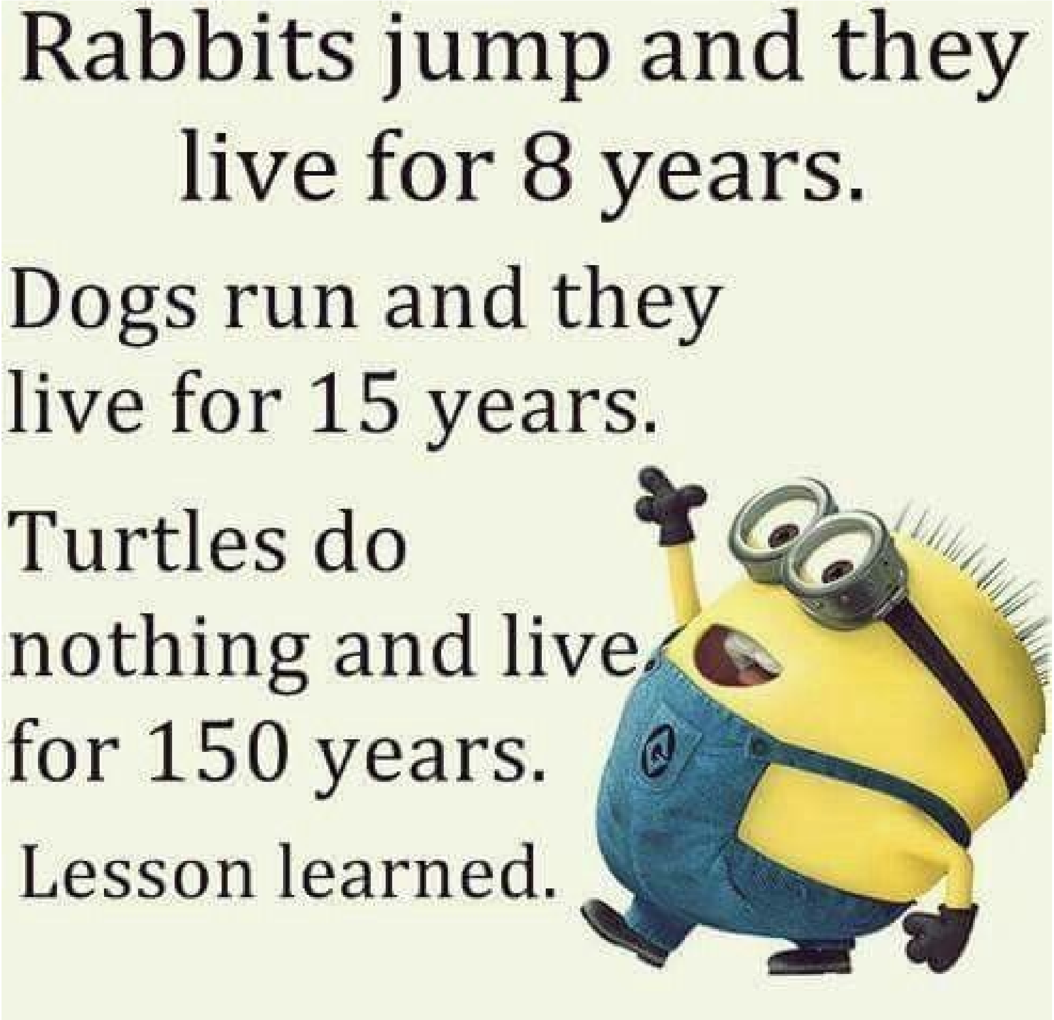
September 29 2017

Pieter Koopman

Radboud University

assignment 2

GENERICS 1



review questions 1

```
:: UNIT = UNIT
instance == UNIT where
   (==) ? ? = ?
```

- do we need a pattern match, or alternatives?

- there is only one option for objects of type UNIT, hence

```
instance == UNIT where
   (==) _ _ = True
```

  - we use UNIT only to satisfy the type system for objects of kind *→* without actual arguments, it contains no information

- the only snag is lazy evaluation,
  using patterns forces evaluation while variables doesn't

review question 2


```
:: CONS a = CONS String a

instance == (CONS a) | == a where
  (==) (CONS _ x) (CONS _ y) = x == y
```

or

```
instance == (CONS a) | == a where
  (==) (CONS a x) (CONS b y) = a == b && x
== y
```

- the generic structure with `EITHER`s guarantees that we have identical constructors,
  there is **no** reason to check the name of the constructor!

review question 3

```
:: Bin   a = Leaf | Bin (Bin a) a (Bin a)
:: BinG  a :== EITHER (CONS UNIT)
                (CONS (PAIR (Bin a) (PAIR a (Bin
a))))
:: ListG a :== EITHER (CONS UNIT)
                (CONS (PAIR a [a]))

fromList []   → LEFT (CONS "Nil" UNIT)
fromBin Leaf  → LEFT (CONS "Leaf" UNIT)
```

we just decided not to look at these strings

```
gEq [] Leaf → True ?
```

- no, this is a type error

generic serialization

version without generic information

```
instance serialize UNIT where
  write UNIT c = c
  read _ l = Just (UNIT, l)

instance serialize (PAIR a b)
      | serialize a & serialize b where
  write (PAIR a b) c = write a (write b c)
  read l = case read l of
    Just (a, m) = case read m of
      Just (b, n) = Just (PAIR a b, n)
      _ = Nothing
    _ = Nothing
```

generic serialization

```
instance serialize (EITHER a b)
      | serialize a & serialize b where
  write (LEFT  a) c = write a c
  write (RIGHT b) c = write b c
  read l = case read l of
    Just (a,m) = Just (LEFT a,m)
    _ = case read l of
      Just (b,m) = Just (RIGHT b,m)
      _ = Nothing
```

generic serialization

```
instance serialize (CONS a) | serialize a
where
  write (CONS s a) c = ["(",s:write a
[")":c]]
  read ["(",s:l] = case read l of
    Just (a,[")":m]) = Just (CONS s a, m)
    _ = Nothing
  read _ = Nothing
```

generic serialization

```
instance serialize [a] | serialize a where
  write l c = write (fromList l) c
  read l = case read l of
    Just (g, m) = Just (toList g, m)
    _ = Nothing

instance serialize (Bin a) | serialize a
where
  write a c = write (fromBin a) c
  read l = case read l of
    Just (a, m) = Just (toBin a, m)
    _ = Nothing
```

same conversions as in gEq, gMap, ..

prettier serialization

`["Leaf"]` instead of `["(","Leaf",")"]`

```
instance serialize (CONS a) | serialize a
where
  write (CONS s a) c = ["(",s:write a
[")":c]]
  read ["(",s:l] = case read l of
    Just (a,[")":m]) = Just (CONS s a, m)
    _ = Nothing
  read _ = Nothing
```

- brackets only if `a` is not `UNIT`
- Clean does not allow separate instances of serialize for `CONS UNIT` and `CONS a`
- solution: make a new class

checking if type is UNIT

```
class isUNIT a :: a → Bool
instance isUNIT UNIT where isUNIT _ = True
instance isUNIT a    where isUNIT _ = False
```

default, matches any type

```
instance serialize (CONS a)
        | serialize, isUNIT a where
  write (CONS s a) c | isUNIT a
   = [s:c]
   = ["(",s:write a [")":c]]
  read ....
```

did we forgot `write a` ?

reflection: is this foolproof?

```
:: Coin = Head | Tail
:: CoinG :== EITHER (CONS UNIT) (CONS UNIT)

instance serialize Coin where
  write coin c = write (fromCoin coin) c
  read list = case read list of
    Just (g,l) = Just (toCoin g,l)
    _          = Nothing
```

- why does this fail?
- solutions:
  1. check names in `toCoin`
  2. check names in `read`

too late for backtracking

read with checking names

- equip the read with a tree of constructor names

```
class serialize a where
  write :: a [String] → [String]
  readB :: B [String] → Maybe (a,[String])
:: B = C String | B B B | N
read = readB N
```

- `write` is unchanged
- many read's ignore the B completely
  - only when it is necessary to check names, or to manipulate the tree of names the tree B is actually used

constructing tree of names

```
instance serialize Coin where
  write coin cont = write (fromCoin coin) cont
  readB _ l
   = case readB (B (C "Head") (C "Tail")) l of
     Just (g,l) = Just (toCoin g,l)
     _ = Nothing
instance serialize [a] | serialize a where
  write l c = write (fromList l) c
  readB _ l
   = case readB (B (C ”Nil”) (C ”Cons”)) l of
     Just (g,m) = Just (toList g,m)
     _ = Nothing
```

checking names

```
instance serialize (CONS a) | serialize a
where
  write ...
  readB (C n) [”(”,s:l]
    = case read l of
        Just (a, [”)”:m]) = Just (CONS s a,m)
        _ = Nothing
  readB _      l = Nothing
```

why don’t we need a B here?

checking names and handle UNIT special

```
instance serialize (CONS a)
        | serialize, isUNIT, readC a where
  write ...
  readB (C b) l = readC read b l
  readB _     l = Nothing

:: READ a :== [String] → Maybe (a,[String])

class readC a | serialize a where
   readC :: (READ a) String [String]
                                 → Maybe (CONS a,
[String])
```

reading constructors

```
instance readC UNIT where
  readC _ n [a: x] | n == a
    = Just (CONS n UNIT, x)
    = Nothing
  readC _ _ _ = Nothing

instance readC default where
  readC f n ["(", a: x] | n == a
    = case f x of
        Just (b,[")":y]) = Just (CONS n b,y)
        _ = Nothing
    = Nothing
  readC _ _ _ = Nothing
```

distributing names

```
instance serialize (EITHER a b)
  | serialize, readC a & serialize, readC b where
  write ...
  readB (B x y) l = case readB x l of
    Just (a,m) = Just (LEFT a,m)
    _ = case readB y l of
      Just (b,m) = Just (RIGHT b,m)
      _ = Nothing
  readB _ _ = Nothing

instance serialize (PAIR a b)
  | serialize a & serialize b where
  write (PAIR a b) c = write a (write b c)
  readB n l = case read l of
    Just (a,m) = case read m of
      Just (b,n) = Just (PAIR a b,n)
      _ = Nothing
    _ = Nothing
```

why don't we need a B here?

assignment 3

GENERICS 2

kind indexed generic programming

- the approach used here fails for type constructor classes
  - like `map`, `Container`, ..
  - due to the different kinds the class based approach does not work
- using a class for each kind solves this problem
  - pass the manipulation for arguments as argument
  - the class system can no longer find the right instance
- the kind indexed approach works also for ordinary classes
  - hence we always use this kind indexed approach
- the generic representation does not change
  - only its manipulations

kind indexed serialization

often called `serialize0`, `write0` and `read0`

- for kind * we still use

```
class serialize a | isUNIT a where
  write :: a [String] → [String]
  read  :: [String] → Maybe (a,[String])
```

- for kind *→* we define

```
:: Write a :== a [String] → [String]
:: Read a  :== [String] → Maybe (a,[String])

class serialize1 t where
 write1 :: (Write a) (t a) [String] → [String]
 read1  :: (Read a) [String]→Maybe (t a,
[String])
```

- similar for other kinds needed

use of argument functions

- e.g. for kind $* \to * \to *$
  - the kind dictates that there are two function arguments
  - instance for

```
:: EITHER a b = LEFT a | RIGHT b
```

  - use arguments instead of ‘recursive’ calls to the class

```
instance serialize2 EITHER where
  write2 wa wb (LEFT  a) c = wa a c
  write2 wa wb (RIGHT b) c = wb b c
  read2 ra rb l = case ra l of ...
```

checking constructor names

- the argument functions can do this!

```
class serializeCONS a where
   writeCons :: (Write a) (CONS a) [String] → [String]
   readCons  :: String (Read  a) [String]
                                 → Maybe (CONS a, [String])
```

- when we give `readCons` a constructor name it has the type `READ a`

```
instance serializeCONS UNIT where
   writeCons wa (CONS s a) c = [s
   readCons n ra [s: l] | n == s
      = Just (CONS s UNIT, l)
   readCons _ _ _ = Nothing
```

why don't we use `wa`

using `serializeCons`

- in the instance of `serialize` for actual datatypes we know the constructor names, specify them!

```
instance serialize Coin where
 write c s = ...
 read  l
  = case read2 (readCons "Head" read)
               (readCons "Tail" read) l of ...

instance serialize1 [] where
 write1 writea l s = ...
 read1  reada l
  = case read2 (readCons ”Nil” read)
               (readCons ”Cons”
                 (read2 reada (read1 reada))) l of
   ...
```

:: CoinG :== EITHER (CONS UNIT) (CONS UNIT)

:: ListG a :== EITHER (CONS UNIT) (CONS (PAIR a [a]))

requirements

1. avoid generic information in the serialized form
   - no `LEFT, RIGHT, PAIR, UNIT, CONS, ..`
2. use only brackets around a constructor with all its arguments
   - no brackets if there are no arguments
   - e.g. `["(","Bin"," ","Leaf"," ","True"," ","Leaf",")”]`
   - with Basic types only: `(Bin Leaf True Leaf)`
3. make sure that you implementation passes all tests
   - feel free to add tests
   - include output as a comment in your program

native generics in Clean

- `import StdGeneric` imports generic types, set compiler flag
- define generic functions one-by-one

```
generic write a :: a [String] → [String]

class serialize a | read{|*|}, write{|*|} a
```

- make instances for basic types and generic types `UNIT, EITHER, PAIR, CONS, OBJECT`
- there is an additional argument for each → in the kind

```
write{|UNIT|}             unit         c = c
write{|OBJECT|} wa        (OBJECT a)   c = wa a c
write{|PAIR|}   wa wb     (PAIR a b)   c = ...
```

*

→

→→*

- derive it for the types needed

```
derive write [], Bin, Coin
```

indicate kind in applications

- indicate the kind in applications of generic functions

```
import GenEq
```

as defined in the lecture

```
Start =
   (gEq{|*|} [1,2] [3,4]
   ,[1,2] === [3,4]
   ,gEq{|*->*|} (==) [1,2] [3,4]
   )
```

- this produces `(False,False,False)`

additional info from Clean

- OBJECT indicates the type of generic objects
- Clean provides information about objects and constructors

```
:: GenericConsDescriptor =
   { gcd_name     :: String              // name of constructor
   , gcd_arity    :: Int                 // arity of constructor
   , gcd_prio     :: GenConsPrio                // priority and associativity
   , gcd_type_def :: GenericTypeDefDescriptor // type def of constructor
   , gcd_type     :: GenType                  // type of the constructor
   , gcd_fields   :: [GenericFieldDescriptor] // non-empty for records
   , gcd_index    :: Int                      // index in the type def
   }
```

- we can use this name in `write` and `read`:

```
write{|CONS of {gcd_name,gcd_arity}|} wa (CONS a) c
= ..
read{|CONS of {gcd_name}|} ra [s:l] | s == gcd_name
   = ..
   = ..
```

no need to pass the names around

the system knows the constructor we are going to read!

master these generics !

- generics are a key supporting technique used in many places in the rest of this course
  - the ideal tool for lazy programmers: define a new operation for basic types derive it for any new data type used

- make sure you can implement and use generics!
  - note that this has nothing to do with generics in OO, that is just overloading