advanced programming tutorial 2

September 22 2017

Pieter Koopman

Radboud University

assignment 1

OVERLOADING

(de)serialize

```
class serialize a where
   write :: a [String] -> [String]
   read  :: [String] -> Maybe (a, [String])
```

result

remaining input

```
:: Maybe a = Just a | Nothing
```

- `Maybe` result of `read`, it can fail
- do we need

```
class serialize a where
   write :: a [String] -> [String]
   read  :: [String] -> (Maybe a, [String]) ?
```

NO

instance for [a]

```
instance serialize [a] | serialize a where
   write [] c = [NilString: c]
   write [a:x] c
     = ["(",ConsString: write a (write x [")":c])]
   read [NilString:r]       = Just ([],r)
   read ["(",ConsString:r] =
   case read r of
   Just (a,s) = case read s of Just (x,[")":t]) =
Just ([a:x],t) _ = Nothing
   _ = Nothing
   read _ = Nothing
NilString  :== "Nil"
ConsString :== "Cons"
```

instance for [a] alternative ?

```
instance serialize [a] | toString a where
 write [] c = [NilString: c]
 write [a:x] c
  = ["(",ConsString, toString: a (write x
[")":c])]
 read ...
```

- is this equally good?
- compiler accepts it
- consider `write [[1],[2,3]]`
  the list `[1]` is transformed by `toString` instead of `write`!
- **this is undesirable**

continuations

- the second argument of write is a continuation
  it tell what must be written after this element

```
write [] c = [NilString: c]
write [a:x] c
 = ["(",ConsString: write a (write x [")":c])]
```

- without continuation we would have

```
write :: [a] -> [string] | write a
write [] = [NilString]
write [a:x]
 = ["(",ConsString: write a] ++ write x ++ [")"]
```

linear in the size of the first argument

linear in the size of the first argument

continuations alternative ?

```
instance serialize [a] | serialize a where
 write [] c = [NilString: c]
 write [a:x] c
  = ["(",ConsString: write a []] ++
     write x [")”] ++ c
 read   ...
```

linear in the size of the first argument

- in general there is nothing wrong with `++`
  but recursive calls on long list can be much work

serialize Bin

```
:: Bin a = Leaf | Bin (Bin a) a (Bin a)
instance serialize (Bin a) | serialize a where
  write Leaf c = [”Leaf”: c]
  write (Bin l a r) c
  = ["(",”Bin”: write l (write a (write r [")":c]))]
  read [LeafString:r] = Just (Leaf,r)
  read ["(",BinString:r] =
    case read r of
      Just (l,s) = case read s of
        Just (a,t) = case read t of
          Just (r,[")":u]) = Just (Bin l a r,u)
          _ = Nothing
        _ = Nothing
      _ = Nothing
  read _ = Nothing
```

boring and error prone

kinds

```
:: Bin  a  = Leaf | Bin (Bin a) a (Bin a)
:: Tree a b = Tip a | Node (Tree a b) b (Tree a b)
:: Rose a  = Rose a [Rose a]
:: T1 a b   = C11 (a b) | C12 b
:: T2 a b c = C2 (a (T1 b c))
:: T3 a b c = C3 (a b c)
:: T4 a b c = C4 (a (b c))
```

Bool	*
Bin	*→*
Rose	*→*
Bin Int	*
Tree	*→*→*
T1	(*→*)→*→*
T2	(*→*)→(*→*)→*→*
T3	(*→*→*)→*→*→*
T4	(*→*)→(*→*)→*→*

polykinds

```
:: T3 a b c = C3 (a b c)
```
kind: (*→*→*)→*→*→*

```
:: Box t u = Box (t u)
```
kind: (*→*)→*→*

- application

```
c3a :: T3 (,) Int Int
c3a = C3 (1,2)
```

- polykinds: use * as a kind variable

```
c3b = C3 (Box [1,2,3])
```

  the actual kind is: ((*→*)→*→*)→(*→*)→*→*

  read the kinds as: (a→b→*)→a→b→*

- in Clean: accepted; derived type `c3b :: T3 Box [] Int`
- specifying the type yields `error: conflicting kinds *->* and *`
  - definition without type is accepted by change

polykinds 2

```
:: T4 a b c = C4 (a (b c))
```
kind: (*→*)→(*→*)→*→*

```
:: IntBox t = IntBox (t Int)
```
kind: (*→*)→*

- application

```
c4a :: T4 [] [] Int
c4a = C4 [[1]]
```

- polykinds: use * as a kind variable

```
c4b = C4 (IntBox (Box [5]))
```

  the actual kind is: ((*→*)→*)→((*→*)→*)→(*→*)→*

  read the kinds as: (a→*)→(b→a)→b→*

- in Clean: accepted; derived type `c4b :: T4 IntBox Box []`
- specifying the type yields `error: conflicting kinds *->* and *`
  - different algorithms for checking and deriving kinds

type constructor class: container

```
class Container t where
   Cinsert   :: a (t a) -> t a        | <
   a
   Ccontains :: a (t a) -> Bool       | <, Eq
   a
   Cshow     ::   (t a) -> [String] |
toString a
   Cnew      :: t a
```

- `t` is of kind $* \rightarrow *$
- hence we need instances like `[]` and `Bin`

list as container

```
instance Container [] where
   Cinsert   a c = [a: c]
   Ccontains a c = isMember a c
   Cnew          = []
   Cshow       c = ["{": showElems c ["}"]]
```

using a continuation

a is no type argument of this instance, we cannot impose the restriction here

```
showElems :: [a] [String]->[String] | toString a
showElems []      c = c
showElems [x]     c = [toString x: c]
showElems [x: xs] c
 = [toString x,",":showElems xs c]
```

search trees as container

```
:: Bin a = Tip | Bin (Tree a) a (Tree a)
instance Container Bin where
  Cinsert a Tip = Bin Tip a Tip
  Cinsert a (Bin l b r)
   | a<b = Bin (Cinsert a l) b r
         = Bin l b (Cinsert a r)
  Ccontains a Tip = False
  Ccontains a (Bin l b r)
   | a<b = Ccontains a l
   | b>a = Ccontains b r
         = True
  Cnew   = Tip
  Cshow t = ["{":showElems (TreetoList t []) ["}"]]

TreetoList Tip         c = c
TreetoList (Bin l a r) c = TreetoList l [a: TreetoList r c]
```

O(log N)

O(log N)

assignment 2

GENERICS 1

serialize using generics

- serialization using classes is boring and error prone
- plan:
  - transform any data type to generics
  - serialize the generic representation
  - read the generic representation
  - transform generic representation to data type

- generic types used

```
:: UNIT         = UNIT
:: EITHER a b = LEFT a | RIGHT b
:: PAIR   a b = PAIR a b
:: CONS   a   = CONS String a
```

making generic representations

- we do nothing smart for basic types
  `Int`, `Bool`, `Char`, `Real` ..
  - we ignore records, arrays, special lists etc. for the moment

- for an algebraic data type

1. introduce `EITHER`s to separate the constructors
   - n `EITHER`s for n+1 constructors
2. every alternative starts with a `CONS`
3. use `PAIR`s to glue arguments together
   - n `PAIR`s for n+1 arguments
   - do not transform the arguments!
   - no arguments: use a `UNIT`

we can replace a single argument by `(PAIR a UNIT)`

```
:: Li  = Ni | Ci Int Li
:: LiG :== EITHER (CONS UNIT) (CONS (PAIR (Int Li)))
```

generic representation

- there is choice in generic representations

```
:: D3 = D3 Int Int Int
:: D3G1 :== CONS (PAIR Int (PAIR Int Int))
:: D3G2 :== CONS (PAIR (PAIR Int Int) Int)

:: Val = Int Int | Bool Bool | Char Char
:: ValG1
  :== EITHER (EITHER (CONS Int) (Cons Bool))
             (Cons Char)
:: ValG2
  :== EITHER (CONS Int)
             (EITHER (Cons Bool) (Cons Char))
```

both work equally well, but be consistent

serialization for generic types

```
instance serialize UNIT where
  write _ c          = ["UNIT": c]
  read ["UNIT": c] = Just (UNIT, c)
  read _             = Nothing
```

- do we need to write:

```
write UNIT c         = ["UNIT": c]
```

- similar for the other generic types

generic generalization

- idea: transform [7] to

```
RIGHT (CONS ”Cons” (PAIR 7 (LEFT (CONS ”Nil” UNIT))))
```

- using

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

- this is printed like

```
[“RIGHT”,”(”,”CONS”,“Cons”,”(”,”PAIR”,”7”,”(”,”LEFT”
,”(”,”CONS”,”Nil”,”(”,”UNIT”,”)”,”)”,”)”,”)”,”)”]
```

- reading such a list is quite easy for the function `read`
  - the strings ”`LEFT`” and ”`RIGHT`” tell exactly what to do if there is any choice (in the case for `EITHER`)
  - the class mechanism will select the appropriate instance!

prettier printing

- instead of

```
["RIGHT","(","CONS","Cons","(","PAIR","7","(","LEFT
","(","CONS","Nil","(","UNIT",")",")",")",")",")"]
```

we might prefer the representation

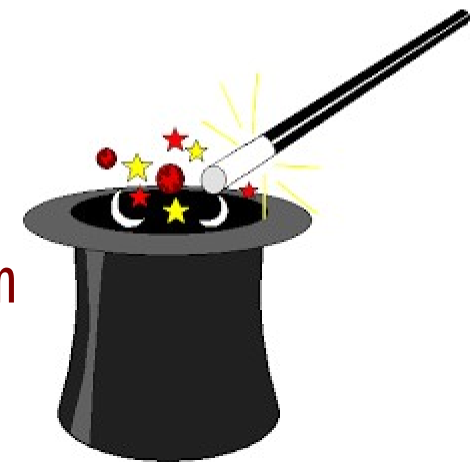
```
["(","Cons","(","7","(","(","Nil",")",")",")"]
```

- in the `write` this is quite easy
- for the `read` we need to backtrack
  - we cannot decide based on `"LEFT"` and `"RIGHT"`
  - just try `LEFT` and if the arguments of the constructor are not there try the `RIGHT` branch
- for `CONS` we just assume that the string in the input is the constructor name

reading with backtracking

- something like (as part of a class `serialize`)

```
read :: [String] -> Either a b | read a &
read b
read list = case read list of
    Just (a,m) = Just (LEFT a, m)
    Nothing    = case read list of
      Just (b,m) = Just (RIGHT b, m
      Nothing    = ..
```



- yes, this looks like magic
  based on the types the compiler selects the right `read`

even prettier printing

- after these changes the `[]` will be printed as
`["(","Nil",")"]`
the result `["Nil"]` is nicer
- but, for `[7]` we want
`["(","Cons","7","Nil",")"]`

- how do we detect this in the generic representation?

- can we do a pattern match? like

```
write (CONS n UNIT) c = [n:c]
write (CONS n a) c = ["(",n: write a [")":c]]
```

if not, how can this be done?