advanced programming
tutorial 5

October 13, 2017

Pieter Koopman

Radboud University

assignment 4

APPETIZER

applicative read student

```
f3 :: IO Student
f3
  =   pure (\_ f _ l _ n.{fname=f,lname=l,snum=n})
  <*> write "Your first name please: "
  <*> read
  <*> write "Your last name please: "
  <*> read
  <*> write "Your student number please: "
  <*> read
```

- or

```
f3
  =   (\_ f _ l _ n.{fname=f,lname=l,snum=n})
  <$> write "Your first name please: "
  <*> ..
```

applicative read student 2: fixing arguments

```
f4 :: IO Student
f4
  =   pure (\f l n.{fname=f, lname=l,
snum=n})
  <*> (write "Your first name please: "
  >>| read)
  <*> (write "Your last name please: "
  >>| read)
  <*> (write "Your student nmber please: "
  >>| read)
```

but this uses the monadic bind…

applicative read student 2: fixing arguments

```
f5 :: IO Student
f5 =
   (\f l n.{fname = f, lname = l, snum = n})
   <$> (oii
   <$> write "Your first name please: "
   <*> read)
   <*> (oii
   <$> write "Your last name please: "
   <*> read)
   <*> (oii
   <$> write "Your student nmber please: "
   <*> read)
   where oii o i = i
```

assignment 4

SERIALIZE

efficient read and write

- the problem

```
:: Serialized :== [String]

read [a:x] = (Just a, x)
read []    = (Nothing, [])

write a l = (UNIT, l ++ [toString a])
```

- write is O(N) in length of the list
- building the list in reverse order makes read O(N)
- needed: read and write in any order in O(1) per element

efficient read and write: solution

```
:: Serialized =
   { ins :: [String]
   , out :: [String]
   }

wrt :: a → State Serialized UNIT| toString a
wrt a = S \s.(Just UNIT,{s & out=[toString a:
s.out]})

rd :: State Serialized String
rd = S r where
   r s=:{ins=[a:x]}     = (Just a,{s & ins = x})
   r s=:{ins=[],out=[]} = (Nothing, s)
   r    {ins=[],out=l}  = {ins=reverse l, out =
[]}
```

O(1)

O(N) for N elements,
O(1) per element

matching and predicate

```
match :: a → Serialize a | toString a
match a
  =   rd
  >>= \s.guard (toString a == s)
  >>| pure a

pred :: (String→Bool) → Serialize String
pred f = rd >>= \s.guard (f s) >>| pure s
```

- `match` with `pred`?

```
match2 :: a → Serialize String | toString a
match2 a = pred ((==) (toString a))

match3 :: a → Serialize a | toString a
match3 a = pred ((==) (toString a)) >>| pure
a
```

State and Functor

```
:: State s a = S (s → (Maybe a, s))

unS :: (State s a) → s → (Maybe a, s)
unS (S f) = f

instance Functor (State s) where
  fmap :: (a→b) (State s a) → State s b
  fmap f (S g) = S \s.case g s of
     (Just a, s) = (Just (f a), s)
     (_,      s) = (Nothing   , s)
```

applicative

```
:: State s a = S (s → (Maybe a, s))

instance Applicative (State s) where
  pure :: a → State s a
  pure a = S \s.(Just a,s)
  (<*>).. :: (State s (a→b)) (State a)→State
s b
  (<*>) (S f) (S x) = S \s.case f s of
    (Just f, s) = case x s of
      (Just x, s) = (Just (f x), s)
      (_     , s) = (Nothing, s)
    (_,      s) = (Nothing, s)
```

monad

```
:: State s a = S (s → (Maybe a, s))

instance Monad (State s) where
  bind :: (State s a) (a→State s b) → State s
b
  bind (S a) f = S \s.case a s of
     (Just a, s) = unS (f a) s
     (_,      s) = (Nothing, s)
```

fail and OrMonad

```
:: State s a = S (s → (Maybe a, s))

instance fail (State s) where
  fail :: State s a
  fail = S \s.(Nothing, s)

instance OrMonad (State s) where
  (<|>)..:: (State s a) (State s a) → State
s a
  (<|>) (S f) (S g) = S \s.case f s of
    (Nothing, _) = g s
    other = other
```

the original state !!

State cannot be unique

other monadic stuff

- there is a bunch of other operators and functions:

  `<$>, >>=, >>|, rtrn, guard`

- in the library `monad.dcl` they are all defined by macro's in terms of the things defined above, e.g.

```
class fail m | Applicative m where
   fail :: m a
   guard :: Bool → m a | fail m
   guard b :== if b (pure undef) fail
class Monad m | Applicative m where
   bind :: (m a) (a→m b) → m b
   (>>=) infixl 1 :: (m a) (a→m b) → m b | Monad m
   (>>=) a f :== bind a f
```

- hence these operators and functions can be used for free, nothing has to be defined

serialize with kind index classes

```
class serialize a | isUNIT a where
  write :: a → Serialize String
  read  ::     Serialize a

:: Write a :== a → Serialize String
:: Read a  :== Serialize a

class serialize1 t where
  write1 :: (Write a) (t a) → Serialize String
  read1  :: (Read  a)       → Serialize (t a)

class serialize2 t where
  write2 :: (Write a) (Write b) (t a b) → Serialize String
  read2  :: (Read  a) (Read  b)         → Serialize (t a b)

class serializeCONS a where
   writeCons :: (Write a) (CONS a) → Serialize String
   readCons  :: String (Read a)    → Serialize (CONS a)
```

instances kind *

```
instance serialize Bool
  where  write b = wrt b
  read = match True <|> match False

instance serialize Int where
  write i = wrt i
  read = rd >>= \s.pure (toInt s) >>=
         \i.guard (s == toString i) >>| pure
i

instance serialize UNIT where
  write _ = pure ""
  read = pure UNIT
```

look ma: no state seen

instances kind *→*→*

```
instance serialize2 EITHER where
  write2 wa wb (LEFT  a) = wa a
  write2 wa wb (RIGHT b) = wb b
  read2 ra rb = LEFT <$> ra <|> RIGHT <$>
rb

instance serialize2 PAIR where
  write2 wa wb (PAIR a b)
    = wa a >>| wrt " " >>| wb b
  read2 ra rb
    = PAIR <$> ra <*> (match " " >>| rb)
```

instance CONS

```
instance serializeCONS UNIT where
   writeCons wa (CONS name a) = wrt name
   readCons name ra
     = CONS <$> match name <*> pure UNIT

instance serializeCONS a where
  writeCons wa (CONS name a) =
    wrt "(" >>| wrt name >>| wrt " " >>| wa a >>|
    wrt ")”
  readCons name ra =
    match "(" >>| CONS <$> match name <*>
    (match " " >>| ra) >>= \c. match ")" >>| pure
c
```

serialization of lists

```
instance serialize1 [] where
  write1 writea l =
    write2 (writeCons write)
           (writeCons (write2 writea
                              (write1 writea)))
    (fromList l)
  read1 reada =
    toList <$>
    read2 (readCons NilString read)
          (readCons ConsString (read2 reada
                                      (read1
reada)))
```

look ma: no state seen

serialize with native generics: write

```
generic write a :: a → State Serialized String

write{|Bool|} b = wrt b
write{|Int|}  i = wrt i
write{|UNIT|} _ = pure ""
write{|PAIR|} wx wy (PAIR x y)
  = wx x >>| wrt " " >>| wy y
write{|EITHER|} wx wy (LEFT x)  = wx x
write{|EITHER|} wx wy (RIGHT y) = wy y
write{|CONS of {gcd_name,gcd_arity}|} wa (CONS a)
 | gcd_arity == 0
  = wrt gcd_name
  = wrt "(" >>| wrt gcd_name >>| wrt " " >>| wa a >>|
    wrt ")"
write{|OBJECT|} wa (OBJECT a) = wa a
```

look ma: no state seen

avoiding the bug in generic read

```
generic read a::State Serialized a // fails
generic read a::(Serialized→(Maybe a, Serialized))
read{|Bool|} = unS (match True <|> match False)
read{|Int|} =
   unS (rd >>= \s.pure (toInt s)
   >>= \i.guard (s == toString i) >>| pure i)
read{|UNIT|} = unS (pure UNIT)
read{|CONS of {gcd_name,gcd_arity}|} ra | gcd_arity == 0
 = unS (match gcd_name >>| CONS <$> S ra)
 = unS (match "(" >>| match gcd_name >>| match " " >>|
   S ra >>= \a. match ")" >>| rtrn (CONS a))
read{|OBJECT|} ra = unS (OBJECT <$> S ra)
read{|PAIR|} ra rb
  = unS (PAIR <$> S ra <*> (match " " >>| S rb))
read{|EITHER|} ra rb = unS (LEFT <$> S ra <|> RIGHT <$> S rb)
```

applying this

```
derive read  [], Bin, Coin, (,), u
derive write [], Bin, Coin, (,), U
```

assignment 5

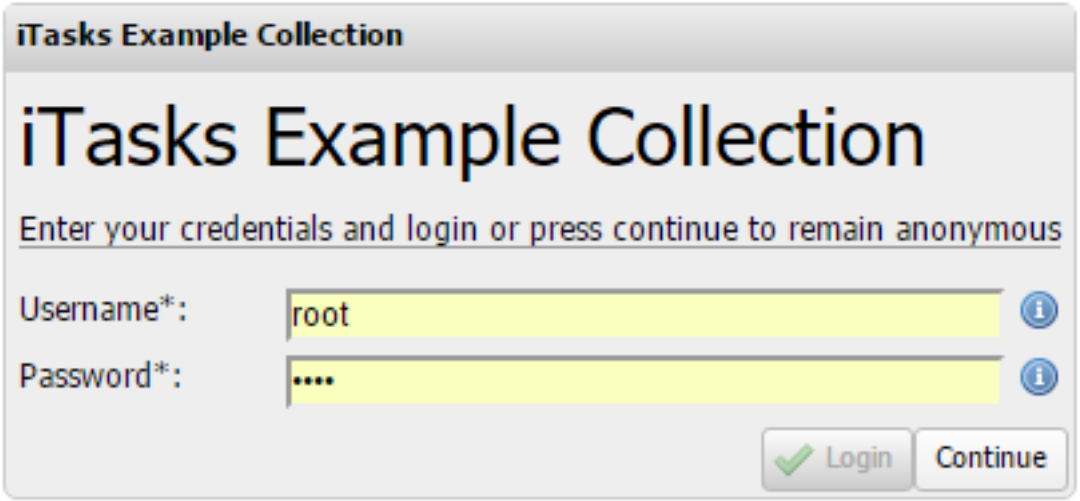
ITASK BASICS

technicalities

- Windows: use a new iTask system from http://clean.cs.ru.nl/Download_Clean
  - use the latest development release with iTasks
  - for Linux and Mac OSX there are similar versions, look at the `README.txt` and `INSTALL.txt`
- iTask is a system under development
  - adding features
  - simplifying the architecture
  - improving the efficiency …
  - this gives issues □ □ □
  - often bugs are removed in version of today

getting started

- use `BasicAPIExamples.icl` as a first test
  - found inside the Clean folder at `iTasks-SDK/Examples`
  - make a project
  - select the `iTasks` environment
- running the program will gives a console that asks you to open a browser window at http://localhost
  - ignore Warnings about SAPL
  - grant the program the requested access
  - use Chrome or Firefox as browser
  - login without giving a name and password, or use `root` and `root`



your own iTask program

- use the iTasks environment
- executable in the `Examples/iTask` or a subfolder
- an appropriate Start rule is:

```
Start :: *World → *World
Start world = startEngine myTask world
```

- a simple task can be

```
myTask :: Task Student
myTask = enterInformation "student" []
```