CSC324 Principles of Programming Languages

Lecture 11

November 27, 2019

Announcements

Exercise 10:

- Very well done: (mean 77%, median 92%)
- A lot more people submitted!
- A number of zeros due to syntax issues

Lab 11 will be time for you to complete A2.

Move assignment 2 deadline to December 4th?

Additional office hours

- Thursday Nov 28, 4:30pm-6pm (James)
- Friday Nov 29, 11am-12pm (Lisa - CSC290 students has priority)
- Monday Dec 2, 12pm-2pm (Lisa)
- Tuesday Dec 3, 1:30pm-3pm (Hassan)
- Tuesday Dec 3, 5:30pm-7pm (James)
- Wednesday Dec 4, 11am-12pm (Lisa)

Grade Distribution (again)

Class average so far is now 70%

	Week 8	Week 11	Change
A	32%	38%	+6%
B	13%	18%	+5%
C	18%	21%	+3%
D	19%	12%	-7%
	18%	10%	-8%

Last Few Classes

- Haskell's Type System
- Generic / Parametric Polymorphism
  - Parametric Types
  - Type Constructors
  - e.g. `[]`, `Maybe`, etc.
- Ad hoc polymorphism
  - Type classes
  - e.g. `Show`, `Eq`, `Ord`, `Functor`, `Foldable`

Today

- Numeric type class
- Higher-order type class: functor
- Chaining failing computations

Afterwards. . .

- Course Evaluations
- Debrief Exercise 10
- Time for Assignment 2

Terminology Review

- Ad hoc polymorphism: function behaviour depends on the *type* of its parameters
- The function `show` is ad hoc polymorphic
- Types can become members of the `Show` type class by implement the function `show`

More Type Classes

```
-- Eq supports == and /=
> :t (==)
(==) :: Eq a => a -> a -> Bool

data Point = Point Float Float

instance Eq Point where
  -- (==) :: Point -> Point -> Bool
  (==) (Point x1 y1) (Point x2 y2) = and [x1 == x2,
                                          y1 == y2]
```

Numbers

```
> a = 5
> b = 10.0
> a + b
15.0
> c = 5 :: Int
> d = 10.0
> c + d
???
```

Numeric Type Classes

- The main numeric class is called `Num`
  - It has an `Integral` subclass for integers
  - It has a `Fractional` subclass for non-integral numbers

```
> :t (*) -- works with any numeric type class
(*) :: Num a => a -> a -> a
> :t (/) -- works only with Fractional type classes
(/) :: Fractional a => a -> a -> a
```

Numeric Literals

```
> :t 1
1 :: Num a => a
> :t 1.0
1.0 :: Fractional p => p
```

- These are polymorphic constants!
- They take on the type required by the type context
  - e.g. in `1 + 2.3`, `1` will be taken as a `Fractional`

A Higher-Order Type Class

```
map :: (a -> b) -> [a] -> [b]
```

There is a function `fmap` that "maps over" many different types, not just lists

```
> fmap (+ 1) (Just 4)
Just 5
> fmap (+ 1) Nothing
Nothing
> fmap even [1, 2, 3]
[False,True,False]
```

Functors

A **functor** is a type class that supports mapping

```
class Functor f where -- f is [], Maybe, etc.
  fmap :: (a -> b) -> f a -> f b
```

- The `a -> b` function does the mapping
- `f a` is the functor type constructor applied to type `a`
  - e.g. `Maybe Int`
- `f b` is the functor type constructor applied to type `b`
  - e.g. `Maybe Int` again, or `Maybe String`, etc.

Example: `List` as Functor

```haskell
data List a = Empty | Cons a (List a) deriving (Show)

instance Functor List where
  -- fmap :: (a -> b) -> List a -> List b
  fmap f Empty        = Empty
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Exercise: `Pair` as Functor

Make `Pair` an instance of `Functor`, where mapping a function over `Pair a` would map the function over each of its values.

```
data Pair a = Pair a a

instance Functor Pair where
  -- fmap :: (a -> b) -> Pair a -> Pair b
```

Exercise: Maybe as a Functor

```haskell
data Maybe a = Nothing | Just a deriving (Show)

instance Functor Maybe where
  -- fmap :: (a -> b) -> Maybe a -> Maybe b
```

Failing Computations

Sequencing Computations

- Suppose we want to perform a sequence of operations, each of which might fail
  - Taking the head/tail of a list
  - Dividing by some number
  - ...
- How might such computations look?

In C

```
int ret;
ret = f1(...);
if (ret == -1)
  ... process error
else {
  ret = f2();
  if (ret == -1)
    ...process error
  else {
    ...
```

In Python

```
try:
  f1(...)
  f2(...)
except ...:
  ...process error
```

Exceptions

In the imperative paradigm, exceptions are “side effects”.

Exceptions in Racket (that won't work)

```
>>> (define (div x y)
        (if (equal? y 0)
            "Error"
            (/ x y)))
>>> (+ (div 4 0) 5) -- won't work
```

Exceptions in Racket using continuations

```
>>> (define (div x y)
        (if (equal? y 0)
            (shift k "Zero Division Error")
            (/ x y)))
>>> (+ (div 4 0) 5) -- will work!
```

Handling Failures in Haskell

In Haskell, we use Maybe to represent failing computation!

The `Maybe` type constructor

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

We can think of `Maybe` as representing two conditions - A success condition `Just` - A failure condition `Nothing`

Safe Functions

We can use `Maybe` to write safe versions of functions Examples:

```
safeHead [] = Nothing
safeHead (x:_) = Just x

safeTail [] = Nothing
safeTail (_:xs) = Just xs
```

Exercise: What are the types of these functions?

How do we **compose** these functions?

Recall lab 10, composeMaybe:

```
composeMaybe :: (a -> Maybe b) -> (b -> Maybe c) ->
                                  (a -> Maybe c)
```

How do we **compose** these functions?

Recall lab 10, composeMaybe:

```
composeMaybe :: (a -> Maybe b) -> (b -> Maybe c) ->
                        (a -> Maybe c)

composeMaybe f g x =
    case (f x) of
        Nothing  -> Nothing
        (Just y) -> g y
```

Composing even more functions...

```
composeMoreMaybe f g h x =
    case (f x) of
        Nothing  -> Nothing
        (Just y) -> case (g y) of
                        Nothing  -> Nothing
                        (Just z) -> h z
```

Composing (Unsafe) Functions

We can use the function composition operation `.` to compose unsafe functions:

```
> add1ToHead = (+ 1) . head
> add1ToHead [10, 20, 30]
11
> add1ToHead [50]
51
> add1ToHead [] -- not safe!
*** Exception: Prelude.head: empty list
```

Composing Safe Functions?

Can we use `.` on safe functions?

```
safeAdd1ToHead = (+ 1) . safeHead
```

But this is a type error!

Wrong Types

These are the types of the components of `(+ 1) . safeHead`

```
safeHead :: [a] -> Maybe a
(+ 1)    :: Float -> Float
```

Problems: we want `(+ 1)` to take `Maybe Float` and return `Maybe Float`.

Lifting Functions

We want to change an `a -> b` function to a `Maybe a -> Maybe b` function

```
lift :: (a -> b) -> (Maybe a -> Maybe b)
lift f = g
         where g Nothing = Nothing
               g (Just x) = Just (f x)
```

This is called **lifting** a function.

Lifting Functions...

There we go:

```
safeAdd1ToHead :: [Float] -> Maybe Float
safeAdd1ToHead = (lift (+ 1)) . safeHead
```

```
> safeAdd1ToHead [1, 2, 3]
Just 2
> safeAdd1ToHead []
Nothing
```

Composing Lifted Functions

If we have only one safe function, we can compose as many functions as we like:

```
(lift (* 2)) . (lift (+ 1)) . safeHead
```

But we're in trouble if we try to compose multiple safe functions!

```
safeHead . safeTail
```

Composing Safe Functions

We would like a function `safeSecond` that returns the second element of a list.

What is the problem with the composition: `safeHead . safeTail`?

Composing Safe Functions

We would like a function `safeSecond` that returns the second element of a list.

What is the problem with the composition: `safeHead . safeTail`?

```
safeTail :: [a] -> Maybe [a]
safeHead :: [a] -> Maybe a
```

The types don't match!

Lifting `safeHead`

Question: why can't we fix this with `lift`?

```
(lift safeHead) . safeTail -- what's wrong?
```

Lifting `safeHead`

Question: why can't we fix this with `lift`?

```
(lift safeHead) . safeTail -- what's wrong?
```

```
safeHead      :: [a] -> Maybe a
lift          :: (c -> d) -> (Maybe c -> Maybe d)
lift safeHead :: Maybe [a] -> Maybe (Maybe a)
```

Return Type of `(lift safeHead) . safeTail`

Unfortunately, `(lift safeHead)` returns a `Maybe (Maybe a)` instead of a `Maybe a`!

```
safeSecond :: [a] -> Maybe (Maybe a)
```

Elementary Implementation of `safeSecond`

Let's start with a correct implementation with no fancy higher-order functions.

```
safeSecond xs =
    let xs' = safeTail xs in
        case xs' of
            Nothing -> Nothing
            Just xs'' -> safeHead xs''
```

The function `safeHead` is correctly called on a List, not a Maybe.

safeThird‘

Exercise: implement `safeThird` using a similar technique.

`safeFourth`?

Factoring Out andThen

```haskell
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just x) f = f x
```

Implementing `safeSecond` Using `andThen`

```
safeSecond xs = andThen (safeTail xs) safeHead
```

Implementing `safeSecond` Using `andThen`

```
safeSecond xs = andThen (safeTail xs) safeHead
```

But this is hard to read. Binary functions can be made infix:

```
> div 8 2
4
> 8 `div` 2
4
```

Implementing `safeSecond` Using `andThen`

```
safeSecond xs = andThen (safeTail xs) safeHead
```

But this is hard to read. Binary functions can be made infix:

```
> div 8 2
4
> 8 `div` 2
4
```

Final version:

```
safeSecond xs = safeTail xs `andThen` safeHead
```

Preparing for Graduate School

Professional vs Research Masters

Professional Masters:

- Take more courses, possibly an internship or research project
- **Goal**: To get a job that requires an advanced degree.

Research Masters:

- Take some courses
- Mostly conduct your own (publishable) research
- **Goal**: To conduct research, maybe start a PhD

Graduate School

To successfully apply to graduate school, you will need:

- Good grades
- Letters of reference from three referees
  - i.e. letter from profs who **knows you**
  - A letter that simply says that you did well in their course is not helpful for the admissions committee.

What can you do?

Get to know the profs whose courses are aligned with your interests:

- Have conversations about the research area in office hours.
- Participate in course message boards.
- Participate in extra-curricular activities.
- Explore doing a reading course or independent studies course with someone
  - If you are doing well in this course and want to do a reading course next term in PL or ML, let me know!

Don't wait until your fourth year to do this!

Exercise 10

Converting split to CPS

```
split :: [Int] -> ([Int], [Int])
split [] = ([], [])
split (x:[]) = ([x], [])
split (x:y:rest) = let (xs, ys) = split rest
                   in ((x:xs), (y:ys))
```

Conversion to cpsSplit

```
cpsSplit :: [Int] -> (([Int], [Int]) -> r) -> r
cpsSplit []     k = k ([], [])
cpsSplit (x:[]) k = k ([x], [])
cpsSplit (x:y:rest) k =
    cpsSplit rest (\(xs, ys) -> k ((x:xs), (y:ys)))
```

Converting merge to CPS

```
mergeSort :: [Int] -> [Int]
mergeSort []  = []
mergeSort [x] = [x]
mergeSort lst = let (xs, ys) = split lst
                    xs' = mergeSort xs
                    ys' = mergeSort ys
                in merge xs' ys'
```

Conversion to cpsMerge

```
cpsMergeSort :: [Int] -> ([Int] -> r) -> r
cpsMergeSort []  k = k []
cpsMergeSort [x] k = k [x]
cpsMergeSort lst k =
    cpsSplit lst (\(xs, ys) ->
    cpsMergeSort xs (\xs' ->
    cpsMergeSort ys (\ys' ->
    cpsMerge xs' ys' k)))
```

Converting eval to CPS: builtins

```
eval env (Plus a b)  = case ((eval env a), (eval env b)) of
    (Num x, Num y) -> Num (x + y)
    _              -> Error "plus"

cpsEval env (Plus a b) k =
    cpsEval env a (\va ->
    cpsEval env b (\vb ->
        case (va, vb) of
            (Num x, Num y) -> k (Num (x + y))
            _              -> Error "plus"))
```

Converting eval to CPS: if

```
eval env (If cond expr alt) = if (eval env cond) == T
    then (eval env expr)
    else (eval env alt)


cpsEval env (If cond t f) k =
    cpsEval env cond (\vc ->
        if vc == T
            then (cpsEval env t k)
            else (cpsEval env f k))
```

Converting eval to CPS: app

```
cpsEval env (App proc args) k =
    cpsEval env proc (\vproc ->
    cpsArg  env args [] (\vargs ->
        case vproc of
            Procedure (Proc fn) -> fn vargs k
            _ -> Error "app"))
; helper
cpsArg env []         vargs k = k (reverse vargs)
cpsArg env (arg:args) vargs k =
    cpsEval env arg (\varg ->
    cpsArg env args (varg:vargs) k)
```

Assignment 2

Warmup: Transform this code in Bork into CPS

```
facEnv = def [
  ("fac",
   Lambda ["n"]
    (If (Equal (Var "n") (Literal (Num 0)))
        (Literal (Num 1))
        (Times (Var "n") (App (Var "fac")
           [(Plus (Var "n") (Literal (Num (-1))))])))))]
```