CSC324 Principles of Programming Languages

Lecture 10

November 20, 2019

Announcements

Additional TA office hours (by James)

- Tuesday Nov 19, 5:30pm-7pm
- Thursday Nov 21, 4:30pm-6pm
- Tuesday Nov 26, 5:30pm-7pm
- Thursday Nov 28, 4:30pm-6pm
- Tuesday Dec 3, 5:30pm-7pm

Today:

- Review the types terminology from last class
- Type classes!
- Discuss Exercises 9 and 10
- Time to work on Exercise 10 / Assignment 2

Review

Haskell Types

What do the following type signatures mean?

```
f :: Int -> Int -> Int

g :: [(Int -> Bool) -> Bool]

h :: Int -> (Int -> Bool)
```

Haskell Types

What do the following type signatures mean?

```
f :: Int -> Int -> Int
```

```
g :: [(Int -> Bool) -> Bool]
```

```
h :: Int -> (Int -> Bool)
```

The Haskell REPL command `:t` is *not* a Haskell expression!

Creating Type Aliases

```
type TypeEnv = Map.Map String Type
```

Here, `TypeEnv` is just a shorthand for `Map.Map String Type`.

The `type` keyword does *not* create a new type!

Creating New Algebraic Data Types

```
data Point = MyPoint Float Float

data Shape = Circle Point Float
           | Rectangle Point Point
```

What are the names of the new *types*?

What are the names of the *value constructors* for each type?

Types of Constructors

```
data Point = MyPoint Float Float

data Shape = Circle Point Float
           | Rectangle Point Point
```

What are the *types* of each of the value constructors?

Pattern Matching

We can pattern match on *value constructors*:

```
area :: Shape -> Float
area (Circle     _ radius) = ...
area (Rectangle (MyPoint x1 y1) (MyPoint x2 y2)) = ...
```

This type of *pattern matching* uses what's called *value destructors*

Polymorphic Types

What do the following type signatures mean?

```
f :: a -> b -> a
```

```
g :: [(Int -> b) -> b]
```

```
h :: a -> Maybe a
```

Maybe

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

Match the Haskell construct on the left with the appropriate programming languages terminology on the right:

Haskell	PL Term
Maybe	Type
Nothing	Type constructor
Maybe Int	Value constructor
a	Type Variable
Just	

Why Types?

- Type Checking Help Prevent Bugs
- Constraining types of polymorphic functions constrains their implementations!

Generic Polymorphism: Constraints (1)

```
f :: a -> a
```

What are the possible implementations for `f`?

(Remember that `a` can be instantiated to *any* type!)

Generic Polymorphism: Constraints (2)

```
f :: a -> b -> a
```

What are the possible implementations for `f`?

Generic Polymorphism: Constraints (3)

```
f :: a -> [a]
```

What are the possible implementations for `f`?

Generic Polymorphism: Constraints (4)

```
f :: [a] -> [a]
```

What are the possible implementations for `f`?

Generic Polymorphism: Constraints (5)

```
f :: a -> b
```

What are the possible implementations for `f`?

Impure Functions

Why do we not discuss these type constraints in imperative languages?

```
T f(T x) {
    deleteFiles();
    return x;
}
```

Type Classes and Ad Hoc Polymorphism

Type Classes

- A **type class** defines one or more functions
- A type can be made a member of the type class by implementing the functions for that type
- Think about a type class as similar to a Java *interface*; types *implement* the interface by implementing the required functions

The Show Type Class

```
> 5
5
> "hi"
"hi"
> data Point = Point Float Float
> Point 3 4
error: No instance for (Show Point)
```

Point is not a member of the type class Show!

Automatically Deriving

Remember the "deriving Show" in a type declaration that we ignored earlier?

```
> data Point = Point Float Float deriving Show
> Point 3 4
Point 3.0 4.0
```

- This gives us a default way to show our values
- But using `deriving` doesn't give us control over how our values are shown.

The Show Type Class: Definition

```
class Show a where
    show :: a -> String
```

So, if we implement function show for a type, then that type is a member of type class Show.

Using `instance`

Writing our own definition of `show`:

```
instance Show Point where
  show (Point x y) =
     "(" ++ (show x) ++ ", " ++ (show y) ++ ")"
```

The `show` Function

The function `show` is interesting because it has multiple implementations, one for each instance of `Show`!

```
> :t show
show :: Show a => a -> String
```

- The `Show a` is a **type class constraint**
- It means that `a` is required to be a type that is an instance of `Show`
- `a` is called a **constrained type variable**

Ad Hoc Polymorphism

- A function is **ad hoc polymorphic** if its behaviour depends on the type of its parameters
- e.g. `show` is ad hoc polymorphic: what it does depends on the type of its parameter

Ad Hoc Polymorphism in Java

Method overloading

```
public int f(int n) {
    return n + 1;
}

public void f(double n) { // different return type
    System.out.println(n);
}

public String f(int n, int m) { // two parameters
    return String.format("%d %d", n, m);
}
```

More Type Classes

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

-- Ord supports <, <=, >, >=
> :t (<)
(<) :: Ord a => a -> a -> Bool

-- Read supports read, which converts from strings
> :t read
read :: Read a => String -> a
> read "5" :: Int
5
```

Exercises

Exercise 9

- Very well done (average 75% median 100%)
- Several submissions were not runnable
- One person has a medical extension so I can't debrief today
- *But*, I will post a sketch of a solution in the next two days on Quercus

Exercise 10

For Task 2 the code for handling Lambda definitions are provided to you in the new handout.

Let's talk more about CPS today and how to approach the problem.

Continuation Passing Style

Why CPS?

- To build an interpreter that supports continuation
- Then we can add constructs like exceptions, backtracking, generators, etc

Golden Rule of CPS

No procedure is allowed to return to its caller—ever.

. . . or. . .

Procedures takes a *callback* to invoke upon their return value.

(from http://matt.might.net/articles/by-example-continuation-passing-style/
)

Example: Identity Function

```
id :: a -> a
id x = x
```

Add a continuation *k*:

```
cps_id :: a -> (a -> r) -> r
cps_id x k = k x
```

(Think of *k* as the *callback* or *return* function)

Example: Insertion

```
insert :: [Int] -> Int -> [Int]
insert []     y = [y]
insert (x:xs) y = if x > y
    then y:x:xs
    else x:(insert xs y)
```

Add a continuation k:

```
cpsInsert :: [Int] -> Int -> ([Int] -> r) -> r
cpsInsert []     y k = k [y]
cpsInsert (x:xs) y k = if x > y
    then k (y:x:xs)
    else cpsInsert xs y (\res -> k (x:res))
```

Rules to CPS a function (1)

1. To CPS a variable, apply k to the variable

```
id     x   = x
cps_id x k = k x
```

Rules to CPS a function (2)

2. To CPS a function call, move the continuation of the function call into its last argument:

```
f     x   = 1 + (g x)
cps_f x k = cps_g x (\result -> k (1 + result))
```

. . . because the continuation of `(g x)` in the expression `1 + (g x)` is `\result -> 1 + result`!

Rules to CPS a function (2)

2. To CPS a function call, move the continuation of the function call into its last argument:

```
f     x   = 1 + (g x)
cps_f x k = cps_g x (\result -> k (1 + result))
```

. . . because the continuation of `(g x)` in the expression `1 + (g x)` is `\result -> 1 + result`!

Another example:

```
f     x   = (g x) + (h x)
cps_f x k = cps_g x (\gx ->
              cps_h x (\hx ->
              k (gx + hx)))
```

Rules to CPS an if statement (3)

3. To CPS an if statement, first CPS the condition. Then, CPS the "then" and "else" branches separately.

```
f     x   = if (g x) then (h x) else (j x)
cps_f x k = cps_g x (\gx ->
              if gx
              then (cps_h x k)
              else (cps_j x k))
```

Example: Function Arguments

```
f      x    = (g (h x) (j x))
cps_f x k = cps_h x (\hx ->
              cps_j x (\jx ->
              cps_g hx jx k))
```

Let's CPS this together

```
fac n = if n == 0
        then 1
        else n * (fac (n - 1))
```