CSC324 Principles of Programming Languages

Lecture 10

November 23/24, 2020

Reminders

- Test 4 is this Tuesday during the lab
  - Lisa is away part of the day
  - If you run into issues, email BOTH Lisa and Andi
- Exercise 10 is posted
  - There's a bit more flexibilty on ex10
  - Email Lisa if there are issues with the deadline (we can afford to be a bit more lenient)
- Next week is Project 2 week: all lectures become office hours
- Check the exam schedule for the exam date/time

Ex9 Debrief

Logic programming is always a challenging part of the course.

CSC324 requires you to think upside-down.

Logic programming requires you to think inside-out.

Ex9 Task 2

```
(define (changeo coins total denoms)
  (conde ((== coins '())
          (zeroo total))
         ((fresh (c coins^ subtotal)
            (== coins (cons c coins^))
            (membero c denoms) ; check if "c" is (car denom
            (pluso subtotal c total)
            (changeo coins^ subtotal denoms)))))
```

Ex9 Task 2

```
(define (changeo coins total denoms)
  (conde ((== coins '())
          (zeroo total))
         ((fresh (c coins^ subtotal)
            (== coins (cons c coins^))
            (== denoms (cons d denoms^)) ; get first/rest 
            (membero c denoms)
            (pluso subtotal c total)
            (changeo coins^ subtotal denoms)))))
```

Ex9 Task 2

```
(define (changeo coins total denoms)
  (conde ((== coins '())
          (zeroo total))
         ((fresh (c coins^ subtotal d denoms^)
            (== coins (cons c coins^))
            (== denoms (cons d denoms^))
            (conde
              ((== c d)
               ; ...
              )
              ((=/= c d)
               ; ...
              ))))))
```

Ex9 Task 2

```
(define (changeo coins total denoms)
  (conde ((== coins '())
          (zeroo total))
         ((fresh (c coins^ subtotal d denoms^)
            (== coins (cons c coins^))
            (== denoms (cons d denoms^))
            (conde
              ((== c d)
               ; ...
              )
              ((=/= c d)
               (changeo coins total denoms^)))))))
;                            ----- -----
```

Ex9 Task 3

Example: we want this call...

```
(pbe ('x 'x) ('y 'x))
```

... to macro expand to ...

```
(run 1
    (body)
    (evalo (list (list 'lambda '(arg) body) '(quote x))
           '()
           'x)
    (evalo (list (list 'lambda '(arg) body) '(quote y))
           '()
           'y))
```

Review

Haskell Types

What do the following type signatures mean?

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

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

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

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 is the type of `Circle`? `Rectangle`?

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]
```

Type Constructor

Type Constructors

Recall that a list is **not** a type:

- A list of `Bool` is a type, a list of `Char` is a type, etc.

A list is a **type constructor** that requires one parameter in order to produce a type

The Type Constructor Maybe

- `Maybe` is another example of a type constructor
- Like lists, `Maybe` requires one parameter in order to produce a type
- Its two value constructors are `Nothing` and `Just`

```
> :t Nothing
Nothing :: Maybe a
> :t Just
Just :: a -> Maybe a
> Nothing
Nothing
> Just 5
Just 5
> if even 5 then Just 5 else Nothing
Nothing -- what is Nothing's type here?
```

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	

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 that returns `Just` some value
- A failure condition that returns `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
```

Q: What are the types of these functions?

Exercise: Types with Maybes

What is a suitable type for `mystery`?

```
mystery (Just _) = Just True
mystery Nothing = Just False
```

Exercise: More Types with Maybes

What is a suitable type for `mystery2`?

```
mystery5 f Nothing = Nothing
mystery5 f (Just x) = Just (f x)
```

Defining Type Constructors

To define a type constructor, use a type variable in a `data` type declaration.

```
data BTree a = Stump | BTree a (BTree a) (BTree a)
```

- first `BTree` is a **type constructor**
- second `BTree` is a **value constructor**
- third/fourth `(BTree a)` are types

Polymorphism in Java

```
class ArrayList<T> { // generic in type T
  ...
}

public static void main(String[] args) {
  ArrayList<Integer> ints = new ArrayList<Integer>();
}
```

Typeclasses and Ad Hoc Polymorphism

Typeclasses

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

The Show Typeclass

```
> 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 typeclass 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` Typeclass: Definition

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

So, if we implement function `show` for a type, then that type is a member of typeclass `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 **typeclass constraint**
- It means that `a` is required to be a type that is an instance of `Show`
- In this case, `a` is 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

```java
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 Typeclasses

```
-- 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
```

Numbers

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

Numeric Typeclasses

- 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 typeclass
(*) :: Num a => a -> a -> a
> :t (/) -- works only with Fractional typeclasses
(/) :: 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`

Higher-Oorder Typeclasses and Functors

A Higher-Order Typeclass

```
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
```

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!