Abstraction++

COS 441 Slides 09

Agenda

- Last week
  - Defining and using type classes
  - Proofs about type classes
    - Induction on the structure of types
  - Case study: A domain-specific language for animation
- This time:
  - More abstraction: Higher-order type classes
  - Kinds: Types for Types

HIGHER-ORDER TYPE CLASSES

A Map For All

- We can map over lists:

```
map f [ ] = [ ]
map f (x:xs) = f x : map f xs
```

A Map For All

- We can map over lists:

```
map f [ ] = [ ]
map f (x:xs) = f x : map f xs
```

- We can map over trees:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

treemap f (Leaf x) = Leaf (f x)
treemap f (Branch l r) = Branch (treemap f l) (treemap f r)
```

A Map For All

- We can map over lists:

```
map f [ ] = [ ]
map f (x:xs) = f x : map f xs
```

- We can map over trees:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

treemap f (Leaf x) = Leaf (f x)
treemap f (Branch l r) = Branch (treemap f l) (treemap f r)
```

- Intuitively, we map over any container data structure.

A Map For All

- We can map over Maybe types:

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

A Map For All

- We can map over Maybe types:

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

- Maps over Maybes can be useful for error handling:

```
type Filename = String

readfile          :: Filename -> IO (Maybe String)
toUpperString :: String -> String


echo = do
   s <- readfile “myfile.txt”
   return (mmap toUpperString s)
```

A Map For All

- We can map over Maybe types:

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

- Maps over Maybes can be useful for error handling:

```
type Filename = String

readfile           :: Filename -> IO (Maybe String)
toUpperString :: String -> String
```

```
echo = do
   s <- readfile "myfile.txt"
   return (mmap toUpperString s)
```

```
echo = do
   s <- readfile "myfile.txt"
   return (
     case s of
       Nothing -> Nothing
       Just x      -> Just (toUpperString x)
   )
```

IO as a Container

- We can think of actions with type IO a as containers as well
  - containers that hold a computation producing a value of type a

```
iomap :: (a -> b) -> IO a -> IO b
iomap f io = do
  x <- io
  return (f x)
```

IO as a Container

- We can think of actions with type IO a as containers as well
  - containers that hold a computation producing a value of type a

```
iomap :: (a -> b) -> IO a -> IO b
iomap f io = do
  x <- io
  return (f x)
```

- Using iomap:

```
getline :: IO string

main = do
   line <- iomap (intersperse '-' . reverse .  map toUpper) getline
   putStrLn line

$ ./io
hello there
E-R-E-H-T- -O-L-L-E-H
```

Functions as Containers

- Even functions can be considered containers:
  - a function with type c -> a "contains" its result (with type a)

```
funmap :: (a -> b) -> (c -> a) -> (c -> b)

funmap f g = f . g
```

A Map for All

- Any container can (and probably should!) support a map:

```
map      :: (a -> b) -> [a] -> [b]
treemap  :: (a -> b) -> Tree a -> Tree b
mmap     :: (a -> b) -> Maybe a -> Maybe b
iomap    :: (a -> b) -> IO a -> IO b
funmap   :: (a -> b) -> (c -> a) -> (c -> b)

eomap    :: (a -> b) -> Either a -> Either b
bstmap   :: (a -> b) -> BST c a -> BST c b
```

- What's the common structure?

A Map for All

- Any container can (and probably should!) support a map:

```
map      :: (a -> b) -> [a] -> [b]
treemap  :: (a -> b) -> Tree a -> Tree b
mmap     :: (a -> b) -> Maybe a -> Maybe b
iomap    :: (a -> b) -> IO a -> IO b
funmap   :: (a -> b) -> (c -> a) -> (c -> b)

eomap    :: (a -> b) -> Either a -> Either b
bstmap   :: (a -> b) -> BST c a -> BST c b
```

- What's the common structure?

```
fmap     :: (a -> b) -> f a -> f b
```

A Second Viewpoint

- we can also think of fmap as a function “lifts” another Haskell function in to a new domain:
  - the domain of animations:

```
fmap :: (a -> b) -> (Behavior a -> Behavior b)
```

  - the domain of error-processors:

```
fmap :: (a -> b) -> (Maybe a -> Maybe b)
```

  - the domain of music:

```
fmap :: (a -> b) -> (Music a -> Music b)
```

  - moral: a map is an extremely general, reuseable idea

Type Constructors

- What is f?

```
fmap :: (a -> b) -> f a -> f a
```

- f isn't a type – types describe values
  - Int, Bool, [Char], Tree Int … are all types
  - there is no value v such that v :: f
  - there are values v such that v :: f Int

- f is actually a new sort of function
  - a function from types to types
  - such functions are often called *type constructors*

Type Constructors

- What is f?

  fmap :: (a -> b) -> f a -> f a

- Instances of f:
  - f might be Tree
  - f might be Maybe
  - f might be [ ] -- list constructor
  - f might be "(->) c" -- function constructor with arg c

- Since type constructors are quite different from types, some constructions don't make any sense:

  (a -> b) -> IO -> IO vs (a -> b) -> IO a -> IO b

  nonsense, a type constructor used in place of a type

Type Constructors

- What is f?

```
fmap :: (a -> b) -> f a -> f a
```

- We need some discipline to avoid nonsensical uses of types and type constructors
- This discipline is typically called a *kind system*
  - a kind system is like a type system, only “one level up”
  - types describe sets of values (eg: Int describes 1, 2, 3, …)
  - *kinds* describe sets of types!

Type Constructors

- What is f?

```
fmap :: (a -> b) -> f a -> f a
```

- ghci can tell you the kind of a type or type constructor:

```
Prelude> :k Int
Int :: *
Prelude> :k Maybe
Maybe :: * -> *
```

- The kind * describes all types (those things that describe values)
- The kind * -> * describes all functions from types to types
- Overall, kinds are types for types – they describe and constrain the way types are used to ensure there's no nonsense
  - Aside: without kinds, a language of types and type functions will be Turing-complete and type checking will be undecidable

The next question

- Can kinds themselves be classified? Perhaps by super-kinds?
- Yes (though they aren't typically called super-kinds)
- NuPrl (a sophisticated theorem prover) has infinite hierarchy of classifiers!
- But most languages stop at 3 levels (values, types, kinds)
- A few go to 4
- There may be one I've ever heard of with 5
- The space of reasonable type systems with 6+ is probably pretty empty (until you go right up to infinity)

- But, 3 levels is more than enough for this class … and it is sufficient for almost any *programming language* that one wouldn't call a theorem prover

Back to fmap

- Where there is a pattern:

```
treemap  :: (a -> b) -> Tree a -> Tree b
funmap   :: (a -> b) -> (c -> a) -> (c -> b)
mmap     :: (a -> b) -> Maybe a -> Maybe b
…
```

- We should create an abstraction:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- Some instances:

```
instance Functor Tree where
  fmap f (Leaf x)   = Leaf (f x)
  fmap f (Tree l r) = Tree (fmap f l) (fmap f l)

instance Functor ((->) c) where
  fmap f g  = f . g
```

Functor Laws

- The functor laws capture the idea that “all” fmap can do is map a function over each element in a container
- A functor can’t do some funny business on the side

```
id = \x -> x

fmap id = id
```

nothing happens when
we map the identity function
over the elements of a functor

Functor Laws

- The functor laws capture the idea that “all” fmap can do is map a function over each element in a container
- A functor can’t do some funny business on the side

```
id = \x -> x

fmap id = id

fmap (f . g) = fmap f . fmap g
```

nothing happens when we map the identity function over the elements of a functor

fmap “preserves” function composition

function composition “commutes” with fmap

Maybe a Functor

```
instance Functor Maybe where
  fmap f Nothing   = Nothing
  fmap f (Just x)  = Just (f x)
```

LAW 1: fmap id = id

SAME AS: for all x::Maybe a, fmap id x = id x

Proof: ?

Maybe a Functor

```
instance Functor Maybe where
  fmap f Nothing   = Nothing
  fmap f (Just x)  = Just (f x)
```

LAW 1: fmap id = id
SAME AS: for all x::Maybe a, fmap id x = id x

a special case of induction

Proof: By cases on x.
case x = Nothing

fmap id Nothing	(RHS of equation)
= Nothing	(unfold fmap at Maybe a)
= id Nothing	(fold id)

case x = Just y

fmap id (Just y)	(RHS of equation)
= Just (id y)	(unfold fmap at Maybe a)
= Just y	(unfold id)
= id (Just y)	(fold id)

Maybe a Functor

```
instance Functor Maybe where
  fmap f Nothing   = Nothing
  fmap f (Just x)  = Just (f x)
```

LAW 2: fmap (f . g) x = (fmap f . fmap g) x

Proof: By cases on x.

case x = Nothing

…

case x = Just y

…

Exercise!

APPLICATIVE FUNCTORS

Mapping Multi-Argument Functions

- So far, we have used fmap with single-argument functions:

```
  fmap (\x -> x + 1) (Just 7)
= Just 8
```

- What happens if we map multi-argument functions?

```
fmap (+) (Just 7) = Just (7+)
```

- We get a Maybe function
  - recall (7+) is the add7 function
  - so is (+7)
  - if you've read LYAHFGG, you know these are called "sections"

Mapping Multi-Argument Functions

- What can we do with a container of functions?

```
Just (7+)
```

- We can use fmap to compose them with other functions:

```
fmap (\f -> f 3) (Just (7+))
```

Mapping Multi-Argument Functions

- What can we do with a container of functions?

```
Just (7+)
```

- We can use fmap to compose them with other functions:

```
fmap (\f -> f 3) (Just (7+))
```

- What can't we do?

```
x : Maybe Int          f :  Maybe (Int -> Int)
```

- fmap does not help us put them together

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

- We can't *compute* in the domain of Maybes

Computing with Behaviors

- Maybe you don't care about computing in the domain of Maybes?

- What about the domain of Behaviors?

```
ball :: Behavior Region
translate ::  Behavior (Region -> Region)

or:

staticBall :: Region
stretch     :: Behavior (Region -> Region)
```

- It would be nice to use those together.

Applicative Functors

- Applicative functors allow you to "lift" computation in to some new domain

```
class Functor f => Applicative f where
    pure :: a -> f a                      -- lift an normal object into the domain
    <*>   :: f (a -> b) -> f a -> f b     -- compute in the domain
```

Applicative Functors

- Applicative functors allow you to "lift" computation in to some new domain

```
class Functor f => Applicative f where
   pure :: a -> f a                      -- lift an normal object into the domain
   <*>   :: f (a -> b) -> f a -> f b     -- compute in the domain
```

```
ball :: Behavior Region
translate ::  Behavior (Region -> Region)

translatedBall = translate <*> ball
```

Applicative Functors

- Applicative functors allow you to "lift" computation in to some new domain

```
class Functor f => Applicative f where
   pure :: a -> f a                      -- lift an normal object into the domain
   <*>   :: f (a -> b) -> f a -> f b     -- compute in the domain
```

```
ball :: Behavior Region
translate ::  Behavior (Region -> Region)

translatedBall = translate <*> ball
```

```
staticBall :: Region
stretch     :: Behavior (Region -> Region)

stretchedBall =
  stretch <*> pure staticBall
```

Maybe the Applicative Functor

```
instance Applicative Maybe where
    pure x = Maybe x
    f <*> x =
        case (f, x) of
          (Just  f’, Just x’) -> Just (f’ x’)
          (_, _)              -> Nothing
```

Maybe the Applicative Functor

```
instance Applicative Maybe where
  pure x = Maybe x
  f <*> x =
    case (f, x) of
      (Just  f', Just x') -> Just (f' x')
      (_, _)              -> Nothing
```

```
x, y, w :: Maybe String
f       :: Maybe (String -> String -> String)

x = pure “hi ”
y = pure “there”
f = pure (++)
w = f <*> x <*> y
  = Maybe “hi there”
```

Maybe the Applicative Functor

```
instance Applicative Maybe where
  pure x = Maybe x
  f <*> x =
    case (f, x) of
      (Just  f’, Just x’) -> Just (f’ x’)
      (_, _)              -> Nothing
```

```
x, y, w :: Maybe String
f       :: Maybe (String -> String -> String)

x = pure “hi ”
y = pure “there”
f = pure (++)
w = f <*> x <*> y
  = Maybe “hi there”
```

```
x, y, w :: Maybe String
f       :: Maybe (String -> String -> String)

x = pure “hi”
y = Nothing -- an error
f = pure (++)
w = f <*> x <*> y
  = Nothing
```

Applicative Laws

- pure should do nothing but put an element in a container
- <*> should do nothing but apply a function in a container to an object in a container
- the laws:

lifting the identify function has no effect

pure id <*> v = v

<*> is just function application
in the lifted domain

pure f <*> pure x = pure (f x)

pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

u <*> pure y = pure (\f -> f y) <*> u

analogous to:
(u . v) w = u (v w)

applying u to a lifted an argument
== extracting the underlying function from u and applying it to the unlifted argument

Applicative Laws

- pure should do nothing but put an element in a container
- <*> should do nothing but apply a function in a container to an object in a container
- connecting fmap to applicative functors:

fmap f x = pure f <*> x

fmap simply applies a pure function to an encapsulated object

Summary

- Haskell has some very general abstraction mechanisms
  - polymorphic functions like map and foldr can be reused on container data structures (like lists) that contain different sorts of elements
  - type classes make it possible to define one interface to be used over different sorts of containers
- This works out because Haskell has a kind system
  - kinds control the way types and type constructors are used
    - without them, Haskell's type system would be undecidable
  - types control the way values are used
  - * is the kind of types
  - * -> * is the kind of functions (like Maybe) from types to types
- Applicative functors "lift" computation to a new domain
- Read LYAHFGG Chap 7, pg 146 -152, Chap 11
- http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Applicative.html