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
We can map over lists:

\[
\begin{align*}
\text{map } f \left[ \right] &= \left[ \right] \\
\text{map } f \left( x : xs \right) &= f \, x \, : \, \text{map } f \, xs
\end{align*}
\]
A Map For All

- We can map over lists:

  \[
  \text{map } f \; \text{[]} = \text{[]}
  \]

  \[
  \text{map } f \; (x:xs) = f \; x \; : \; \text{map } f \; xs
  \]

- We can map over trees:

  \[
  \text{data Tree } a = \text{Leaf } a \mid \text{Branch } (\text{Tree } a) \; (\text{Tree } a)
  \]

  \[
  \text{treemap } f \; (\text{Leaf } x) = \text{Leaf } (f \; x)
  \]

  \[
  \text{treemap } f \; (\text{Branch } l \; r) = \text{Branch } (\text{treemap } f \; l) \; (\text{treemap } f \; r)
  \]
A Map For All

• We can map over lists:

\[
\text{map } f \left[ \right] = \left[ \right] \\
\text{map } f \left( x:xs \right) = f \left( x \right) : \text{map } f \left( xs \right)
\]

• We can map over trees:

\[
\text{data } \text{Tree } a = \text{Leaf } a \mid \text{Branch } (\text{Tree } a) \left(\text{Tree } a\right)
\]

\[
\text{treemap } f \left( \text{Leaf } x \right) = \text{Leaf } \left( f \left( x \right) \right) \\
\text{treemap } f \left( \text{Branch } l \left( r \right) \right) = \text{Branch } \left( \text{treemap } f \left( l \right) \right) \left( \text{treemap } f \left( r \right) \right)
\]

• Intuitively, we map over any container data structure.
We can map over Maybe types:

```
    mmap f Nothing  = Nothing
    mmap f (Just x) = Just (f x)
```
We can map over Maybe types:

\[
\text{mmap } f \text{ Nothing} = \text{Nothing} \\
\text{mmap } f \text{ (Just } x) = \text{Just } (f \ x)
\]

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)
We can map over Maybe types:

\[
\begin{align*}
\text{mmap } f \text{ Nothing} &= \text{Nothing} \\
\text{mmap } f \text{ (Just } x) &= \text{Just } (f \text{ x})
\end{align*}
\]

Maps over Maybes can be useful for error handling:

type Filename = String

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

echo = do
  s <- readFile "myfile.txt"
  return (mmap toUpperCaseString s)

  case s of
    Nothing -> Nothing
    Just x  -> Just (toUpperCaseString x)
• We can think of actions with type IO a as containers as well
  – containers that hold a computation producing a value of type a

  \[
  \text{iomap} :: (a \rightarrow b) \rightarrow \text{IO } a \rightarrow \text{IO } b \\
  \text{iomap } f \text{ io } = \text{ do} \\
  x \leftarrow \text{ io} \\
  \text{return } (f x)
  \]
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

  \[
  \text{iomap} :: (a \rightarrow b) \rightarrow \text{IO } a \rightarrow \text{IO } b \\
  \text{iomap } f \text{ io } = \text{do} \\
  \hspace{1cm} x \leftarrow \text{io} \\
  \hspace{1cm} \text{return } (f x)
  \]

- Using iomap:

  \[
  \text{getline} :: \text{IO string} \\
  \text{main } = \text{do} \\
  \hspace{1cm} \text{line } \leftarrow \text{iomap } (\text{intersperse } '‐' \cdot \text{reverse } \cdot \text{map } \text{toUpper}) \text{ getline} \\
  \hspace{1cm} \text{putStrLn line}
  \]

  \$ ./io
  hello there
  E-R-E-H-T- -O-L-L-E-H
• Even functions can be considered containers:
  – a function with type \( c \rightarrow a \) “contains” its result (with type \( a \))

\[
\text{funmap} :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (c \rightarrow b)
\]

\[
\text{funmap } f \ g = f \ . \ g
\]
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?
• 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:
    \[
    \text{fmap :: (a -> b) -> (Behavior a -> Behavior b)}
    \]
  - the domain of error-processors:
    \[
    \text{fmap :: (a -> b) -> (Maybe a -> Maybe b)}
    \]
  - the domain of music:
    \[
    \text{fmap :: (a -> b) -> (Music a -> Music b)}
    \]
  - moral: a map is an extremely general, reusable idea
What is f?

- 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?

  \[ \text{fmap} :: (a \to b) \to f\ a \to f\ a \]

• 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:

  \[(a \to b) \to \text{IO} \to \text{IO}\ \ \text{vs}\ \ (a \to b) \to \text{IO} \ \ a \to \text{IO} \ b\]

  nonsense, a type constructor used in place of a type
• What is $f$?

\[
\text{fmap :: (a \rightarrow b) \rightarrow f\; a \rightarrow f\; a}
\]

• We need some discipline to avoid nonsensical uses of types and type constructors

• This discipline is typically called a \textit{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, ...)
  – \textit{kinds} describe sets of types!
Type Constructors

• What is f?

\[ \text{fmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ a \]

• 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
• Where there is a pattern:

\[
\begin{align*}
\text{treemap} & : (a \to b) \to \text{Tree} a \to \text{Tree} b \\
\text{funmap} & : (a \to b) \to (c \to a) \to (c \to b) \\
\text{mmap} & : (a \to b) \to \text{Maybe} a \to \text{Maybe} b \\
& \ldots
\end{align*}
\]

• We should create an abstraction:

\[
\text{class Functor } f \text{ where}
\begin{align*}
fmap & : (a \to b) \to f a \to f b
\end{align*}
\]

• Some instances:

\[
\begin{align*}
\text{instance Functor } \text{Tree} \text{ where}
& \quad \text{fmap } f \text{ (Leaf } x) = \text{Leaf } (f \ x) \\
& \quad \text{fmap } f \text{ (Tree } l \ r) = \text{Tree } (\text{fmap } f \ l) (\text{fmap } f \ l)
\end{align*}
\]

\[
\begin{align*}
\text{instance Functor } ((\to) \ c) \text{ where}
& \quad \text{fmap } f \ g = f \ . \ g
\end{align*}
\]
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

\[ \text{id} = \{x \mapsto x\} \]

\[ \text{fmap id} = \text{id} \]

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 = \( x \to x \)

\[ \text{fmap } id = id \]

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

\[ \text{fmap } (f \cdot g) = \text{fmap } f \cdot \text{fmap } g \]

fmap “preserves” function composition

function composition “commutes” with fmap
LAW 1: \[ \text{fmap } id = id \]
SAME AS: for all \( x :: \text{Maybe } a \), \( \text{fmap } id \ x = id \ x \)

Proof: ?
instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)

LAW 1:  \( \text{fmap id} = \text{id} \)
SAME AS: for all \( x :: \text{Maybe a} \), \( \text{fmap id} \ x = \text{id} \ x \)

Proof: By cases on \( x \).
case \( x = \text{Nothing} \)
    \( \text{fmap id} \ \text{Nothing} \) (RHS of equation)
    = \text{Nothing} (unfold fmap at \( \text{Maybe a} \))
    = \text{id} \ \text{Nothing} (fold id)
case \( x = \text{Just} \ y \)
    \( \text{fmap id} \ \text{Just} \ y \) (RHS of equation)
    = \text{Just} (\text{id} \ y) (unfold fmap at \( \text{Maybe a} \))
    = \text{Just} \ y (unfold \text{id})
    = \text{id} (\text{Just} \ y) (fold id)
instance Functor Maybe where
    fmap f Nothing   = Nothing
    fmap f (Just x)  = Just (f x)

LAW 2:   \( \text{fmap} (f \cdot g) \ x = (\text{fmap} f \cdot \text{fmap} g) \ x \)

Proof: By cases on \( x \).
\[
\begin{align*}
\text{case } x = \text{Nothing} \\
\hspace{1cm} \ldots \\
\text{case } x = \text{Just } y \\
\hspace{1cm} \ldots \\
\end{align*}
\]

Exercise!
APPLICATIVE FUNCTORS
Mapping Multi-Argument Functions

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

\[
\text{fmap } (\lambda x \to x + 1) (\text{Just } 7) = \text{Just } 8
\]

• What happens if we map multi-argument functions?

\[
\text{fmap } (+) (\text{Just } 7) = \text{Just } (7+)
\]

• We get a Maybe function
  – recall (7+) is the add7 function
  – so is (+7)
  – if you’ve read LYAHFFGG, you know these are called “sections”
Mapping Multi-Argument Functions

- What can we do with a container of functions?

  \[
  \text{Just (7+)}
  \]

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

  \[
  \text{fmap (\lambda f \rightarrow f 3) (Just (7+))}
  \]
Mapping Multi-Argument Functions

• What can we do with a container of functions?

  \[
  \text{Just (7+)}
  \]

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

  \[
  \text{fmap } (\lambda f \rightarrow f 3) (\text{Just (7+)})
  \]

• What can’t we do?

  \[
  x : \text{Maybe Int} \quad f : \text{Maybe (Int \rightarrow Int)}
  \]

• fmap does not help us put them together

  \[
  \text{fmap :: (a \rightarrow b) \rightarrow Maybe a \rightarrow Maybe b}
  \]

• 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?

```plaintext
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

```haskell
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 into some new domain

```haskell
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

```haskell
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
```

```haskell
ball :: Behavior Region
translate :: Behavior (Region -> Region)
translatedBall = translate <*> ball

staticBall :: Region
stretch :: Behavior (Region -> Region)
stretchedBall = stretch <*> pure staticBall
```
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

```haskell
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 = 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:
  - pure id <*> v = v
  - pure f <*> pure x = pure (f x)
  - pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
  - u <*> pure y = pure (\f -> f y) <*> u

Lifting the identify function has no effect

<*> is just function application in the lifted domain

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 uplifted 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:

  \[
  \text{fmap } f \ x = \text{pure } f \ <*> \ x
  \]

  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