PROGRAMMING IN HASKELL

Equational Reasoning and Induction

Equ ning

Functional Programming

- What is functional programming? Some possible answers:
  - Programming with first-class functions

    - map (\x - ~> [2,3,4]

  - Programming with mathe ctions
    - No side-effects (no global mutable state, no IO)
    - Calling a function with the same arguments, always returns the same output (not true in most languages!)

Reasoning about Purely Functional Programs

- When programs behave as mathematical functions, standard mathematical techniques can be used to reason about such programs.

- Such techniqu

  - Equational reasoning: Inte rams as equations; substitute equ ls

  - Structural induction: The use of recursion means that reasoning techniques such as induction are useful.

Equational reasoning in mathematics

- Whenever we have a system of mathematical equations, we can use equational reasoning to reason about such equations. For example:

$x = y + z$
$y = 3z$
$z = 5$

- Suppose we want to find the value of x

Equational reasoning in mathematics

- Using annotated steps we proceed as follows

  $x = y + z$

  $\equiv \{?\}$

Proving that Option is a Monad

- Using equational reasoning and structural induction we can show that the Option instance


```
instance Monad
   return x
   None >>= f       = None
   (Some x) >>= f = f x
```


satisfies the monad laws:

```
return a >>= k  =  k a
m >>= return  =  m
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h
```

Proving that Option is a Monad

- First law:

return a >>= k
≡{?}

Proving that Option is a Monad

- Second law:

m >>= return

≡{?}

Proving that Option is a Monad

- Third law:

m >>= (\x -> k x >>= h)

≡{case analysis (

1) Case m of No

None >>= (\x -> k x >>= h)

≡{?}

Proving that Option is a Monad

- Third law:

m >>= (\x -> k x >>= h)

≡{case analysis (

2) Case m of So

Some a >>= (\x -> k x >>= h

≡{?}

Str ion

Induction in mathematics

Induction decomposes a proof into two parts:

- Base case(s): Prove that the property holds for the base cases.
- Inductive step(s): Prove that ty holds for the recursive cases.

Induction in mathematics

The simplest and most common type of induction is induction on natural numbers.

- Base case: Sh lds for $n = 0$.

- Inductive step: Assuming tha rty holds for n, show that the property holds

In the inductive step, the assumption is called the Induction Hypothesis.

Structural Induction

In functional programming, we can use induction to reason about functions defined over datatypes. For example, given the list datatype:


```
data [a] = [
```


we obtain the following inducti :

- Base case: Show that the property holds for xs = [].
- Inductive step: Assuming that the property holds for xs, show that the property holds for (x:xs).

Structural Induction

- Consider the map function:

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs)   = f
```


id x = x

- It should be clear that mappi tity function returns the same list back:

map id xs ≡ xs

Can we prove it?

Equational reasoning in mathematics

map id xs

≡{?}

Equational reasoning in mathematics

map id xs
≡{?}
2)

Equational reasoning in mathematics

- Consider the map function again:

```
map :: (a -> b) -> [a] -> [b]
map f []        = [
map f (x:xs)  = f
```



```
(f . g) x = f (g x)
```

- Do you think the following is true?

map f (map g xs) ≡ map (f . g) xs — map fusion

Can we prove it?

Equational reasoning in mathematics

map f (map g xs)

≡{?}

1)

Equational reasoning in mathematics

map f (map g xs)
≡{?}
2)

Functors

It turns out that the map function, together with the laws:

map f (map g xs) ≡ map (f . g) xs -- map fusion

map id xs ≡ xs -- map identity

Can be generalized:


```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
   — Laws
   — fmap f (fmap g fa) ≡ fmap (f . g) fa
   — fmap id fa ≡ fa
```

List Functor

Given the map function and our two proofs, it is easy to create an instance for Functor:


```
instance Functor
   fmap = map
```

Other Functors

Functors are quite common, nearly all parametrised types (Example: [a], Maybe a, IO a, …) are functors

— data Maybe a


```
instance Functor Maybe where
   -- fmap :: (a -> b) -> Maybe            e b
   fmap f Nothing = Nothing
   fmap f (Just x) = Just (f x)
```

Maybe Functor

fmap id ma
≡{?}
1)

Maybe Functor

fmap f (fmap g fa)
≡{?}

Exercises:

1) Consider the definitions:

```
map :: (a -> b) -> [a] -> [b]
map f []       =
map f (x:xs)   =

length :: [a] -> Int
length []       = 0
length (x:xs) = 1 + length xs
```


Prove that:

length (map f xs) $\equiv$ length xs

length (map f xs)
≡ {Induction on xs}
1) Base case: xs = []
length (map f [])
≡ {definition of map}
length []
2) Inductive case: xs = (y:ys)
length (map f (y:
≡ {definition of m
length (f y : map f ys)
≡ {definition of length}
1 + length (map f ys)
≡ {Induction Hypothesis}
1 + length ys
≡ {definition of length}
length (y:ys)

Exercises:

2) Consider the definitions:

```
map :: (a -> b) -> [a] -> [b]
map f []        =
map f (x:xs)  =

(++) :: [a] -> [a] -> [a]
[] ++ ys         = ys
(x:xs) ++ ys  = x : (xs ++ ys)
```


Prove that:

map f (xs ++ ys) ≡ map f xs ++ map f ys

map f (xs ++ ys)

≣ {Induction on xs}

1) Case xs = []

map f ([] ++ ys)

≣ {definition of ++}

map f ys

≣ {definition of +

[] ++ map f ys

≣ {definition of map}

map f [] ++ map f ys

Exercises:

3) Consider the definitions:

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

mapT :: (a -> b)
mapT f Leaf          = Leaf
mapT f (Fork x l r) = Fork (f x)              (mapT f r)

flatten :: Tree a -> [a]
flatten Leaf         = []
flatten (Fork x l r)  = x : flatten l ++ flatten r
```

Exercises:

3) Prove that:

mapT id ≡ id

mapT f (mapT g

flatten . mapT f ≡ map f . flatte