PROGRAMMING IN HASKELL

Equational Reasoning and Induction

Equational Reasoning

Functional Programming

- What is functional programming? Some possible answers:
  - Programming with first-class functions
    - map (\x -> x + 1) [1,2,3]   ~> [2,3,4]
  - Programming with mathematical functions
    - 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 techniques include:
  - Equational reasoning: Interpret programs as equations; substitute equals by equals

  - 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

$x = y + z$

$\equiv$ {definition of y}

$x = 3z + z$

$\equiv$ {simplification}

$x = 4z$

$\equiv$ {definition of z}

$x = 4 * 5$

$\equiv$ {simplification}

$x = 20$

Proving that Option is a Monad

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

```
instance Monad Option where
   return x           = Some 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
≣{definition of return}
Some a >>= k
≣{definition of >>=}
k a
```

Proving that Option is a Monad

- Second law:

m >>= return
≡{by induction on m}
1) Case m is None
None >>= return
≡{definition of >>=}
None
2) Case m is Some a
Some a >>= return
≡{definition of >>=}
return a
≡{definition of return}
Some a

Proving that Option is a Monad

- Third law:

m >>= (\x -> k x >>= h)
≣{case analysis (or induction) on m}
1) Case m of None
None >>= (\x -> k x >>= h)
≣{definition of >>=}
None
≣{definition of >>=}
None >>= h
≣{definition of >>=}
(None >>= k) >>= h

Proving that Option is a Monad

- Third law:

```
m >>= (\x -> k x >>= h)
≡{case analysis (or induction) on m}
2) Case m of Some a
Some a >>= (\x -> k x >>= h)
≡{definition of >>=}
(\x -> k x >>= h) a
≡{simplification}
k a >>= h
≡{definition of >>=}
(Some a >>= k) >>= h
```

Structural Induction

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 the property holds for the recursive cases.

Induction in mathematics

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

- Base case: Show that the property holds for $n = 0$.
- Inductive step: Assuming that the property holds for n, show that the property holds for $n + 1$.

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] = [] | a : [a]
```

we obtain the following inductive principle:

- 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 x : map f xs
```

```
id x = x
```

- It should be clear that mapping the identity function returns the same list back:

map id xs $\equiv$ xs

Can we prove it?

Equational reasoning in mathematics

map id xs

≡{induction on xs}

1) Base Case: xs = []

map id []

≡{definition of map}

[]

Equational reasoning in mathematics

map id xs

≡{induction on xs}

2) Inductive Case: xs = (y:ys)

map id (y:ys)

≡{definition of map}

id y : map id ys

≡{definition of id}

y : map id ys

≡{Induction Hypothesis}

y : ys

Equational reasoning in mathematics

- Consider the map function again:

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

(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)

≡{Induction on xs}

1) Base Case: xs = []

map f (map g [])

≡{definition of map}

map f []

≡{definition of map}

[]

≡{definition of map}

map (f . g) []

Equational reasoning in mathematics

map f (map g xs)
≡{Induction on xs}
2) Inductive Case: xs = (y:ys)
map f (map g (y:ys))
≡{definition of map}
map f (g y : map g ys)
≡{definition of map}
f (g y) : map f (map g ys)
≡{Induction Hypothesis}
f (g y) : map (f . g) ys
≡{definition of .}
(f . g) y : map (f . g) ys
≡{definition of map}
map (f . g) (y:ys)

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 [] where
   fmap = map
```

Other Functors

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

— data Maybe a = Nothing | Just a

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

Maybe Functor

fmap id ma
≡{case analysis on ma}
1) Case ma = Nothing
fmap id Nothing
≡{definition of fmap}
Nothing
2) Case ma = Just x
fmap id (Just x)
≡{definition of fmap}
Just (id x)
≡{definition of id}
Just x

Maybe Functor

fmap f (fmap g fa)

≣{case analysis on fa}

1) Case fa = Nothing

fmap f (fmap g Nothing)

≣{definition of fmap}

fmap f Nothing

≣{definition of fmap}

Nothing

≣{definition of fmap}

fmap (f . g) Nothing

Exercises:

1) Consider the definitions:

```
map :: (a -> b) -> [a] -> [b]
map f []        = []
map f (x:xs)  = f x : map f 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:ys))
≡ {definition of map}
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)  = f x : map f xs

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

Prove that:

map f (xs ++ ys) $\equiv$ 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) -> Tree a -> Tree b
mapT f Leaf           = Leaf
mapT f (Fork x l r) = Fork (f x) (mapT f l) (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 xs) ≡ mapT (f . g) xs

flatten . mapT f ≡ map f . flatten