Exam 3 Comprehensive Study Guide

Monoids

Monoid Class Definition

A **Monoid** is an algebraic structure used to model combining and accumulating values. In Haskell, the class is defined as:

```
class Monoid a where
  mempty :: a                          -- Identity element
  mappend :: a -> a -> a               -- Binary operation
  mconcat :: [a] -> a                  -- Fold operation for a list
  (<>) :: a -> a -> a                  -- Infix version of mappend
  x <> y = mappend x y                 -- Default implementation
```

Monoid Functions

- **`mempty`**: Represents the identity element that does not change other elements when combined.
- **`mappend`** **or** **`<>`**: Combines two elements.
- **`mconcat`**: Reduces a list of elements into a single result using the Monoid operations.

Monoid Laws

Monoids must satisfy these laws:

1. **Identity/Unit Laws**:
    - `x <> mempty = x` (Right identity)

- `mempty <> x = x` (Left identity)

2. **Associativity**:

    - `(x <> y) <> z = x <> (y <> z)`

These laws ensure consistent and predictable behavior when combining elements.

Examples of Monoids

Lists

- **Identity**: `[]` (empty list)
- **Operation**: `(++)` (concatenation)

```
[1,2] <> [3,4] = [1,2,3,4]
[] <> [5] = [5]   -- Identity law
```

Numbers (Sum)

- **Identity**: `Sum 0`
- **Operation**: `(+)`

```
Sum 5 <> Sum 10 = Sum 15
Sum 0 <> Sum 3 = Sum 3   -- Identity law
```

Numbers (Product)

- **Identity**: `Product 1`
- **Operation**: `(*)`

```
Product 4 <> Product 3 = Product 12
Product 1 <> Product 7 = Product 7   -- Identity law
```

Maybe

If the inner type is a Monoid, `Maybe` can also form a Monoid.

- **Identity**: `Nothing`
- **Operation**: Combines inner values if both are `Just`.

```
Just (Sum 3) <> Just (Sum 5) = Just (Sum 8)
Nothing <> Just (Sum 5) = Just (Sum 5)  -- Identity law
```

Monads

Monad Class Definition

Monads are used to chain computations where each step may depend on the result of the previous step. In Haskell, the class is defined as:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b  -- Bind operator
  return :: a -> m a                 -- Wraps a value in the monad
  (>>) :: m a -> m b -> m b          -- Sequential execution
```

Monad Functions

- `>>=` **(bind)**: Chains two computations, passing the result of the first to the second.
- `return`: Injects a value into the monadic context.
- `>>`: Sequences two computations, discarding the result of the first.

Monad Laws

Monads must satisfy these laws:

1. **Identity/Unit Laws**:
   - `return a >>= f = f a` (Left identity)
   - `m >>= return = m` (Right identity)
2. **Associativity**:
   - `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

> **Note**: While Monads are conceptually related to Monoids, in Haskell, an instance of Monad may not necessarily also be an instance of Monoid unless explicitly defined as such.

Do-Notation

Haskell provides `do`-notation for working with Monads, making code more readable. It acts as syntactic sugar for `>>=` and sequencing operations.

Example: Using `do`-notation with IO

```
main = do
  putStrLn "Enter your name:"
  name <- getLine
  putStrLn ("Hello, " ++ name)
```

Examples of Monads

Maybe Monad

Handles computations that may fail.

```
Just 5 >>= (\x -> Just (x + 1)) -- Just 6
Nothing >>= (\x -> Just (x + 1)) -- Nothing
```

List Monad

Represents non-deterministic computations.

```haskell
[1,2] >>= (\x -> [x, x*2]) -- [1,2,2,4]
```

IO Monad

Sequences I/O actions.

```haskell
main = do
  putStrLn "Enter a number:"
  num <- readLn
  print (num * 2)
```

Zippers

Basics

A Zipper is a data structure that allows focused traversal and updates of complex structures like lists or trees. It splits the structure into:

1. **Focus**: The current element.
2. **Context**: The rest of the structure.

List Zipper

Representation:

```haskell
type ListZipper a = ([a], a, [a]) -- (left context, focus, right context)
```

Operations:

1. **Move Focus Right**:

```
([1], 2, [3,4]) -> ([1,2], 3, [4])
```

2. **Modify Focus**:

```
([1], 2, [3,4]) -> ([1], 5, [3,4])
```

Tree Zipper

For binary trees:

```haskell
data Tree a = Leaf a | Node a (Tree a) (Tree a)
type Breadcrumb a = (a, Tree a, Tree a)
type TreeZipper a = (Tree a, [Breadcrumb a])
```

Equational Reasoning

Substitution

1. Identify sub-expressions matching known definitions or results.
2. Replace using equivalences.
3. Simplify step-by-step.

Basic Proof Construction

1. **Two-Column Proof**:
    - Write each step of the proof in two columns: the expression being manipulated and the justification.
2. **Induction**:
    - **Base Case**: Prove the simplest instance.
    - **Inductive Step**: Assume for `n`, prove for `n+1`.

Lazy Evaluation Semantics

Evaluation Forms

1. **Normal Form (NF)**: Fully evaluated expression.
2. **Weak Head Normal Form (WHNF)**: Partially evaluated, stopping at the outermost constructor.

Example

```
-- NF
(1 + 2) -> 3

-- WHNF
(1 + 2) remains unevaluated if context does not demand full evaluation.
```

Lazy Semantics

- Be able to reduce an expression to WHNF to simulate lazy semantics.
- Identify sub-expressions that are never evaluated under lazy semantics.