Interval Sets

In this assignment, you will write a library that provides an efficient implementation of *interval sets*. You will use the `Test.QuickCheck` module to write specifications and to test and debug the library.

This assignment makes use of the `Test.QuickCheck` module, which is installed on the CS Department Linux machines, is automatically installed when using the full installer of the Haskell Platform, and can be manually installed when using the core installer of the Haskell Platform by executing `cabal update ; cabal v2-install --lib QuickCheck`.

Read and follow the course Haskell Style Guide (https://www.cs.rit.edu/~mtf/teaching/20191/psfp/style.html). Especially, look for opportunities to eliminate unnecessary complexity (use anonymous functions, use partial applications, use sectioning, use functionals). Also, look for opportunities to use standard library functions.

Use `IntervalSet.hs`, which defines a type and operations for interval sets, `IntervalSetQCInv.hs`, which defines invariant properties (i.e., properties that assert that invariants on interval sets are preserved by the operations), `IntervalSetQCModel.hs` which defines model-based properties (i.e., properties that assert that the behavior of interval sets can be modeled by the behavior of `Data.Set.Set`), and `IntervalSetQCAlg.hs`, which defines algebraic properties (i.e., properties that assert that the behavior of interval sets should satisfy relations among operations).
Overview

The IntervalSet module provides an efficient implementation of interval sets. The ISet type is defined as follows:

```haskell
newtype ISet a = ISet [(a,a)] deriving (Eq, Typeable)
```

```haskell
invariant :: Integral a => ISet a -> Bool
invariant (ISet is) = aux is
where aux [] = True
      aux [(lo1, hi1)] = lo1 <= hi1
      aux ((lo1, hi1):(lo2, hi2):is) = lo1 <= hi1 &&
                                      hi1 < lo2 &&
                                      not (hi1 `isPred` lo2) &&
                                      aux ((lo2, hi2):is)
```

```haskell
isPred :: Integral a => a -> a -> Bool
x `isPred` y = x == y - 1 && y - 1 < y
```

An interval set represents a set of elements as a list of closed (i.e., inclusive of the endpoints) intervals. For example,

- \( \text{ISet } [] \) represents the empty set.
- \( \text{ISet } [(5,5)] \) represents the set \( \{5\} \).
- \( \text{ISet } [(0,5),(10,15)] \) represents the set \( \{0,1,2,3,4,5,10,11,12,13,14,15\} \).
- \( \text{ISet } [(0,0),(2,2),(4,4),(6,6),(8,8),(10,10)] \) represents the set \( \{0,2,4,6,8,10\} \).

An interval set is most efficient when it represents a set with many elements but few intervals.

Note the invariant that asserts that each interval is maintained in sorted order \( (lo_1 \leq hi_1) \), the intervals in the list of intervals are disjoint and are maintained in sorted order \( (hi_1 < lo_2) \), and the intervals in the list of intervals are separated \( (not (hi_1 `isPred` lo_2)) \) (i.e., there exist elements not in the set between intervals). For example,

- \( \text{ISet } [(1,0)] \) is invalid; the set \( \{0,1\} \) should be represented by \( \text{ISet } [(0,1)] \).
- \( \text{ISet } [(4,5),(0,1)] \) is invalid; the set \( \{0,1,4,5\} \) should be represented by \( \text{ISet } [(0,1),(4,5)] \).
- \( \text{ISet } [(0,1),(2,3),(4,5)] \) is invalid; the set \( \{0,1,2,3,4,5\} \) should be represented by \( \text{ISet } [(0,5)] \).\n
isPred is a function such that \( x `\text{isPred`} y \) returns True if \( x \) is the immediate predecessor of \( y \) and returns False otherwise. The Integral a type-class constraint ensures that there are no values of type a between \( y - 1 \) and \( y \). The \( y - 1 < y \) condition causes isPred to return False when \( y - 1 \) “wraps around” due to finite precision (e.g., \((-128) :: \text{Data.Int.Int8}) \sim 1 \sim 127\).

Figures 1 and 2 summarizes the IntervalSet module’s exported type and operations. The worst-case complexities of the operations are expressed in terms of \( s \), the set’s size (i.e., the number of elements in the set), and \( w \), the set’s width (i.e., the number of intervals in the set).

Note that the operations of the IntervalSet module correspond in name and meaning (although not in worst-case complexity) to operations of the Data.Set module (https://downloads.haskell.org/~ghc/8.6.5/docs/html/libraries/containers-0.6.0.1/Data-Set.html).

In order to facilitate testing of the IntervalSet module’s operations with QuickCheck, the type ISet is made an instance of the type-class Arbitrary. Note that the interval sets generated by this instance are not guaranteed to satisfy the interval-set invariant, but do so with high probability (and the probability increases with the bit-width of the Integral type being generated). You may find it instructive to compare the effectiveness of this interval-set generator to that of the very simple, commented-out interval-set generator, for example, by introducing a bug into an operation and comparing the counterexamples (if any) found when using the two interval-set generators.
• Interval Set Type
  – data ISet a
    An interval set of values a.
    Instances:
    * Eq a => Eq (ISet a)
    * Typeable a => Typeable (ISet a)
    * (Typeable a, Show a) => Show (ISet a)
    * (Integral a, Arbitrary a) => Arbitrary (ISet a)
    * Integral a => Semigroup (ISet a)
    * Integral a => Monoid (ISet a)
  – invariant :: Integral a => ISet a -> Bool
    Test if the interval set is valid. \(O(w)\)

• Query Operations
  – null :: ISet a -> Bool
    Is this the empty set? \(O(1)\)
  – size :: Integral a => ISet a -> Int
    The number of elements in the set. \(O(w)\)
  – width :: ISet a -> Int
    The number of intervals in the set. \(O(w)\)
  – member :: Integral a => a -> ISet a -> Bool
    Is the element in the set? \(O(w)\)
  – notMember :: Integral a => a -> ISet a -> Bool
    Is the element not in the set? \(O(w)\)
  – isSubsetOf :: Integral a => ISet a -> ISet a -> Bool
    Is this a subset? \(O(w_1 + w_2)\)
  – isProperSubsetOf :: Integral a => ISet a -> ISet a -> Bool
    Is this a proper subset (i.e., a subset but not equal)? \(O(w_1 + w_2)\)

• Construction Operations
  – empty :: ISet a
    The empty set. \(O(1)\)
  – singleton :: a -> ISet a
    Create a singleton set. \(O(1)\)
  – insert :: Integral a => a -> ISet a -> ISet a
    Insert an element in a set. \(O(w)\)
  – delete :: Integral a => a -> ISet a -> ISet a
    Delete an element from a set. \(O(w)\)

Figure 1: IntervalSet summary
• Combine Operations
  - `union :: Integral a => ISet a -> ISet a -> ISet a`
    The union of two sets. \(O(w_1 + w_2)\)
  - `unions :: Integral a => [ISet a] -> ISet a`
    The union of a list of sets.
  - `intersection :: Integral a => ISet a -> ISet a -> ISet a`
    The intersection of two sets. \(O(w_1 + w_2)\)
  - `difference :: Integral a => ISet a -> ISet a -> ISet a`
    The difference of two sets. \(O(w_1 + w_2)\)

• Filter Operations
  - `filter :: Integral a => (a -> Bool) -> ISet a -> ISet a`
    Filter all elements that satisfy the predicate. \(O(s)\)
  - `partition :: Integral a => (a -> Bool) -> ISet a -> (ISet a, ISet a)`
    Partition the set into two sets, one with all elements that satisfy the predicate and one with all elements that
    don’t satisfy the predicate. \(O(s)\)
  - `split :: Integral a => a -> ISet a -> (ISet a, ISet a)`
    The expression `split x is` is a pair \((is1, is2)\) where `is1` comprises the elements of `is` less than `x` and `is2`
    comprises the elements of `is` greater than `x`. \(O(w)\).
  - `splitMember :: Integral a => a -> ISet a -> (ISet a, Bool, ISet a)`
    Performs a `split` but also returns whether the pivot element was found in the original set. \(O(w)\)

• Min/Max Operations
  - `maxView :: Integral a => ISet a -> Maybe (a, ISet a)`
    Retrieves the maximal element of the set, and the set stripped of that element, or `Nothing` if passed an empty
    set. \(O(w)\)
  - `minView :: Integral a => ISet a -> Maybe (a, ISet a)`
    Retrieves the minimal element of the set, and the set stripped of that element, or `Nothing` if passed an empty
    set. \(O(1)\)

• Conversion Operations
  - `fromList :: Integral a => [a] -> ISet a`
    Create a set from a list of elements. \(O(n \log n)\)
  - `toList :: Integral a => ISet a -> [a]`
    Convert the set to an ascending list of elements. \(O(s)\)

Figure 2: `IntervalSet` summary (continued)
The four major components of the assignment are:

- (10pts) IntervalSetQCInv – complete the collection of invariant properties
- (15pts) IntervalSetQCModel – complete the collection of model-based properties
- (10pts) IntervalSetQCAlg – write a collection of algebraic properties
- (65pts) IntervalSet – complete the implementation of interval sets

One approach is to work on the components in this order, but you will not be able to “test” the properties until completing the implementation. An alternate approach is to work on the properties and implementation for each operation in turn, being sure to “test” and “debug” the properties and implementation for one operation before moving on to the next.

Note that the main actions in the IntervalSetQCInv, IntervalSetQCModel, and IntervalSetQCAlg modules quickCheck each property with interval sets of Integers, Int8s, and Word8s. There are a number of implementation details that concern “wrap around” and dense interval-sets that will rarely arise with random interval sets of Integers, but are much more likely to arise with random interval sets of Int8s and Word8s. Be sure to use type constraints when copying a counterexample into ghci or a source file.

Finally, note that, within the IntervalSetQCInv, IntervalSetQCModel, and IntervalSetQCAlg modules, the IntervalSet module is imported qualified:

```haskell
import IntervalSet (ISet(..))
import qualified IntervalSet as ISet
```

This is because many operation names (but not the type name) clash with Prelude and Data.Set names. Within these modules, for example, one must use ISet.member to refer to the member operation of the IntervalSet module, but one can use ISet to refer to the ISet type of the IntervalSet module.
(10pts) IntervalSetQCInv.hs

For each operation that returns a result that includes one or more ISets, write a QuickCheck property that asserts that the output ISets satisfy invariant (whenever the input ISets satisfy invariant).

For example:

```hs
prop_inv_empty :: Integral a => {a} ISet a -> Bool
prop_inv_empty is = ISet.invariant (ISet.empty `asType0f` is)

prop_inv_singleton :: Integral a => a -> Bool
prop_inv_singleton x = ISet.invariant (ISet.singleton x)

prop_inv_insert :: Integral a => a -> ISet a -> Property
prop_inv_insert x is = ISet.invariant is ==> ISet.invariant (ISet.insert x is)

prop_inv_delete :: Integral a => a -> ISet a -> Property
prop_inv_delete x is = ISet.invariant is ==> ISet.invariant (ISet.delete x is)
```

Complete the following invariant properties:

- (1pt) prop_inv_union :: Integral a => ISet a -> ISet a -> Property
- (1pt) prop_inv_unions :: Integral a => [ISet a] -> Property
- (1pt) prop_inv_intersection :: Integral a => ISet a -> ISet a -> Property
- (1pt) prop_inv_difference :: Integral a => ISet a -> ISet a -> Property
- (1pt) prop_inv_filter :: Integral a => Fun a Bool -> ISet a -> Property
- (1pt) prop_inv_split :: Integral a => a -> ISet a -> Property
- (1pt) prop_inv_splitMember :: Integral a => a -> ISet a -> Property
- (1pt) prop_inv_maxView :: Integral a => ISet a -> Property
- (1pt) prop_inv_minView :: Integral a => ISet a -> Property
- (1pt) prop_inv_fromList :: Integral a => [a] -> Bool
(30pts) IntervalSetQCModel.hs

For each operation, write a QuickCheck property that asserts that the operation on ISets is correct with respect to the corresponding operation on Data.Set.Sets (whenever the input ISets satisfy invariant). Note that the Data.Set.Set instance of Eq is an equivalence relation and not simply structural equality; hence it is appropriate to use (==) to compare Data.Set.Sets.

For example:

```haskell
abstract :: Integral a => ISet a -> Set a
abstract is = Set.fromList (ISet.toAscList is)

prop_model_empty :: Integral a => ISet a -> Bool
prop_model_empty is = Set.empty == abstract (ISet.empty `asTypeOf` is)

prop_model_singleton :: Integral a => a -> Bool
prop_model_singleton x = Set.singleton x == abstract (ISet.singleton x)

prop_model_insert :: Integral a => a -> ISet a -> Property
prop_model_insert x is = ISet.invariant is ==> Set.insert x (abstract is) == abstract (ISet.insert x is)

prop_model_delete :: Integral a => a -> ISet a -> Property
prop_model_delete x is = ISet.invariant is ==> Set.delete x (abstract is) == abstract (ISet.delete x is)
```

Complete the following model-based properties:

- (1pts) prop_model_size :: Integral a => ISet a -> Property
- (1pts) prop_model_member :: Integral a => a -> ISet a -> Property
- (1pts) prop_model_notMember :: Integral a => a -> ISet a -> Property
- (1pts) prop_model_isSubsetOf :: Integral a => ISet a -> ISet a -> Property
- (1pts) prop_model_isProperSubsetOf :: Integral a => ISet a -> ISet a -> Property
- (1pts) prop_model_union :: Integral a => ISet a -> ISet a -> Property
- (1pts) prop_model_unions :: Integral a => [ISet a] -> Property
- (1pts) prop_model_intersection :: Integral a => ISet a -> ISet a -> Property
- (1pts) prop_model_difference :: Integral a => ISet a -> ISet a -> Property
- (1pts) prop_model_filter :: Integral a => Fun a Bool -> ISet a -> Property
- (1pts) prop_model_split :: Integral a => a -> ISet a -> Property
- (1pts) prop_model_splitMember :: Integral a => a -> ISet a -> Property
- (1pts) prop_model_maxView :: Integral a => ISet a -> Property
- (1pts) prop_model_minView :: Integral a => ISet a -> Property
- (1pts) prop_model_fromList :: Integral a => [a] -> Bool
Write a collection of QuickCheck properties that assert relationships among the operations on ISet.

For example:

{- An element is a member of the interval set after insertion. -}
prop_alg_member_insert :: Integral a => a -> ISet a -> Property
prop_alg_member_insert x is = ISet.invariant is ==> ISet.member x (ISet.insert x is)

{- An element is not a member of the interval set after deletion. -}
prop_alg_notMember_delete :: Integral a => a -> ISet a -> Property
prop_alg_notMember_delete x is = ISet.invariant is ==> ISet.notMember x (ISet.delete x is)

{- An insertion may decrease or increase the width by at most one. -}
prop_alg_width_insert :: Integral a => a -> ISet a -> Property
prop_alg_width_insert x is = ISet.invariant is ==> max 0 (wo - 1) <= wn && wn <= wo + 1
  where wo = ISet.width is
  wn = ISet.width (ISet.insert x is)

{- A deletion may decrease or increase the width by at most one. -}
prop_alg_width_delete :: Integral a => a -> ISet a -> Property
prop_alg_width_delete x is = ISet.invariant is ==> max 0 (wo - 1) <= wn && wn <= wo + 1
  where wo = ISet.width is
  wn = ISet.width (ISet.delete x is)

Write 10 (distinct) algebraic properties:
  • 10× (1pt) prop_alg_XXX :: ... -> Property

For each algebraic specification, write a comment that provides a short, intuitive description of the relationship being asserted. Be sure to add quickChecks of each algebraic specification with interval sets of Integers, Int8s, and Word8s to the main action.
(65pts) IntervalSet.hs

Complete the following operation implementations:

- (5pts) size :: Integral a => ISet a -> Int \(O(w)\)
- (5pts) member :: Integral a => a -> ISet a -> Bool \(O(w)\)
- (5pts) isSubsetOf :: Integral a => ISet a -> ISet a -> Bool \(O(w_1 + w_2)\)
  (Note that using toAscList and member will not achieve the required running time.)
- (10pts) union :: Integral a => ISet a -> ISet a -> ISet a \(O(w_1 + w_2)\)
  An almost correct implementation of union has been provided. Use failing test cases to debug and fix.
  (Note that using toAscList, member, insert, and/or delete will not achieve the required running time.)
- (10pts) intersection :: Integral a => ISet a -> ISet a -> ISet a \(O(w_1 + w_2)\)
  (Note that using toAscList, member, insert, and/or delete will not achieve the required running time.)
- (10pts) difference :: Integral a => ISet a -> ISet a -> ISet a \(O(w_1 + w_2)\)
  (Note that using toAscList, member, insert, and/or delete will not achieve the required running time.)
- (5pts) splitMember :: Integral a => a -> ISet a -> (ISet a, Bool, ISet a) \(O(w)\)
  (Note that using partition will not achieve the required running time.)
- (5pts) maxView :: Integral a => ISet a -> Maybe (a, ISet a) \(O(w)\)
- (5pts) minView :: Integral a => ISet a -> Maybe (a, ISet a) \(O(1)\)
- (5pts) fromList :: Integral a => [a] -> ISet a \(O(n \log n)\)
  (Note that using member, insert, and/or delete will not achieve the required running time.)