nus-cs2030s-2122-s2 / lab7-beetee17 Private

Code Issues Pull requests Actions Projects Security

master lab7-beetee17 / cs2030s / fp / InfiniteList.java Go to file ...
/ <> Jump to

beetee17 Cumulative submission Latest commit e713e55 9 days ago History

2 contributors

307 lines (269 sloc) | 8.31 KB Raw Blame

```java
package cs2030s.fp;

import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;

/**
 * Infinite List is similar to a Stream. It memoizes values and is lazyi
 * in evaluation.
 *
 * @author Brandon (Group 12A)
 */
public class InfiniteList<T> {

  private final Lazy<Maybe<T>> head;
  private final Lazy<InfiniteList<T>> tail;
  private static final InfiniteList<?> SENTINEL = new Sentinel();

  /**
   * A private constructor. We only allow initialisation via
   * generate and iterate.
   *
   */
  private InfiniteList() {
    this.head = null;
    this.tail = null;
  }

  /**
   * A private constructor. We only allow initialisation via
   * generate and iterate.
   *
   * @param head The given head to be wrapped in Lazy.
   * @param tail The given tail to be wrapped in Lazy.
   */
  private InfiniteList(T head, Producer<InfiniteList<T>> tail) {
    this.head = Lazy.of(Maybe.some(head));
    this.tail = Lazy.of(() -> tail.produce());
  }

  /**
   * A private constructor. We only allow initialisation via
   * generate and iterate.
   *
   * @param head The given head to.
   * @param tail The given tail to.
   */
  private InfiniteList(Lazy<Maybe<T>> head, Lazy<InfiniteList<T>> tail) {
    this.head = head;
    this.tail = tail;
  }

  /**
   * Creates an InfiniteList according to a Producer.
   *
   * @param <T> The type of the desired InfiniteList.
   * @param producer The given producer to generate values
   * @return An InfiniteList of generated values.
   */
  public static <T> InfiniteList<T> generate(Producer<T> producer) {
    return new InfiniteList<>(Lazy.of(() -> Maybe.some(producer.produce())),
        Lazy.of(() -> generate(producer)));
  }

  /**
   * Creates an InfiniteList according to an initial value and an iterator.
   *
   * @param <T> The type of the desired InfiniteList.
   * @param seed The first value of the InfiniteList
   * @param next The given iterator to generate values
   * @return An InfiniteList of iterated values.
   */
  public static <T> InfiniteList<T> iterate(T seed, Transformer<T, T> next) {
    return new InfiniteList<>(seed, () -> iterate(next.transform(seed), next)
  }

  /** Returns the head's value of the InfiniteList.
   *
   * @return The head's value.
   */
  public T head() throws NoSuchElementException {
    return this.head.get().orElseGet(() -> this.tail.get().head());
  }

  /** Returns the tail of the InfiniteList.
   *
   * @return The tail of the InfiniteList..
   */
  public InfiniteList<T> tail() throws NoSuchElementException {
    InfiniteList<T> tempTail = this.tail.get();
    if (this.head.get().equals(Maybe.none())) {
      return tempTail.isSentinel() ? sentinel() : tempTail.tail();
    }
    return tempTail;
  }

  /**
   * Lazily applies the given transformation to each element in the list
   * and returns the resulting `InfiniteList`.
   *
   * @param <R> type parameter for resulting `InfiniteList`
   * @param mapper the mapping function to be applied on the list.
   *
   * @return an `InfiniteList` with its elements mapped.
   */
  public <R> InfiniteList<R> map(Transformer<? super T, ? extends R> mapper)
    return new InfiniteList<>(Lazy.of(() -> Maybe.some(mapper.transform(this.
        Lazy.of(() -> this.tail().map(mapper)));
  }

  /**
   * Lazily filters out elements in the list that fail a given
   * BooleanCondition. Marks removed elements as Maybe.none().
   *
   * @param predicate The BooleanCondition to test values with.
   * @return The filtered InfiniteList.
   */
  public InfiniteList<T> filter(BooleanCondition<? super T> predicate) {
    Lazy<Maybe<T>> tempHead = Lazy.of(() -> predicate.test(this.head())
        ? Maybe.some(this.head())
        : Maybe.none());
    return new InfiniteList<>(tempHead, Lazy.of(() -> this.tail().filter(pred
  }

  /** Returns a Sentinel.
   *
   * @param <T> The type of the desired Sentinel.
   * @return A Sentinel of the specified type.
   */
  public static <T> InfiniteList<T> sentinel() {
    @SuppressWarnings("unchecked")
    InfiniteList<T> temp = (InfiniteList<T>) SENTINEL;
    return temp;
  }

  /** Terminates an infinite list into a finite one with at most n elements.
   *
   * @param n The maximum length of the truncated list.
   * @return The truncated list.
   */
  public InfiniteList<T> limit(long n) {
    if (n <= 0) {
      return sentinel();
    }

    Producer<InfiniteList<T>> newTail = () -> this.head.get()
        .map(x -> this.tail.get().limit(n - 1))
        .orElseGet(() -> this.tail.get().limit(n));

    return new InfiniteList<>(this.head, Lazy.of(newTail));
  }

  /**
   * Truncates the list as soon as an element does not satisfy a predicate.
   *
   * @param predicate The BooleanCondition to test elements with
   * @return The truncated list.
   */
  public InfiniteList<T> takeWhile(BooleanCondition<? super T> predicate) {
    Lazy<Boolean> cond = Lazy.of(() -> this.head()).filter(predicate);

    Lazy<Maybe<T>> newHead = Lazy.of(() -> cond.get()
        ? Maybe.some(this.head())
        : Maybe.none());

    return new InfiniteList<T>(
        newHead,
        Lazy.of(() -> cond.get() && predicate.test(this.tail().head())
          ? this.tail().helper(predicate)
          : sentinel())
        );
  }

  /**
   * Helper function for the takeWhile method.
   *
   * @param p The BooleanCondition to test elements with
   * @return An InfiniteList.
   */
  private InfiniteList<T> helper(BooleanCondition<? super T> p) {
    return new InfiniteList<T>(this.head(),
        () -> p.test(this.tail().head())
        ? this.tail().helper(p)
        : sentinel());
  }

  /**
   * Checks if this is an instance of Sentinel.
   *
   * @return  true if the list is an instance of Sentinel, and false otherwis
   */
  public boolean isSentinel() {
    return false;
  }

  /**
   * Reduces the list into a single value.
   *
   * @param <U> The type of the return value
   * @param identity The initial value
   * @param accumulator Combiner to combine two values
   * @return The result of accumulating the list from right to left.
   */
  public <U> U reduce(U identity, Combiner<U, ? super T, U> accumulator) {
    return accumulator.combine(this.tail().reduce(identity, accumulator), thi
  }

  /**
   * Gets the length of the list.
   *
   * @return The length of the list.
   */
  public long count() {
    long v = this.head.get().equals(Maybe.none()) ? 0L : 1L;
    return v + this.tail.get().count();
  }

  /**
   * Collects the elements into a List.
   *
   * @return A list of elements of the InfiniteList.
   */
  public List<T> toList() {
    List<T> ls = new ArrayList<>();
    ls.add(this.head());
    ls.addAll(this.tail().toList());
    return ls;
  }

  /**
   * Returns the String representation of the list.
   *
   * @return Wraps the head and tail in square brackets.
   */
  public String toString() {
    return "[" + this.head + " " + this.tail + "]";
  }

  /**
   * Sentinel represents a list that contains nothing.
   * It is used to mark the end of the list.
   */
  private static class Sentinel extends InfiniteList<Object> {

    /**
     * Constructor for a Sentinel instance.
     */
    Sentinel() {
      super();
    }

    @Override
    public Object head() throws NoSuchElementException {
      throw new java.util.NoSuchElementException();
    }

    @Override
    public InfiniteList<Object> tail() throws NoSuchElementException {
      throw new java.util.NoSuchElementException();
    }

    @Override
    public <R> InfiniteList<R> map(Transformer<? super Object, ? extends R> m
      return InfiniteList.sentinel();
    }

    @Override
    public InfiniteList<Object> filter(BooleanCondition<? super Object> predi
      return InfiniteList.sentinel();
    }

    @Override
    public InfiniteList<Object> limit(long n) {
      return InfiniteList.sentinel();
    }

    @Override
    public List<Object> toList() {
      return List.of();
    }

    @Override
    public InfiniteList<Object> takeWhile(BooleanCondition<? super Object> pr
      return InfiniteList.sentinel();
    }

    @Override
    public <U> U reduce(U identity, Combiner<U, ? super Object, U> accumulato
      return identity;
    }

    @Override
    public long count() {
      return 0;
    }

    @Override
```

```java
    public boolean isSentinel() {
      return true;
    }

    @Override
    public String toString() {
      return "-";
    }
  }
}
```