beetee17 Cumulative submission Latest commit e713e55 5 days ago History

1 contributor

140 lines (127 sloc) | 4.03 KB Raw Blame

```java
package cs2030s.fp;

/**
 * Lazy value is useful for cases where producing the value
 * is expensive, but the value might not eventually be used.
 *
 * @author Brandon (Group 12A)
 */
public class Lazy<T> {
  private Producer<? extends T> producer;
  private Maybe<T> value;

  /**
   * A private constructor to initialise a Lazy object according to the facto
   *
   * @param value The given value
   * @param producer The given producer
   */
  private Lazy(Maybe<T> value, Producer<? extends T> producer) {
    this.value = value;
    this.producer = producer;
  }

  /**
   * Initializes the Lazy object with the given value.
   *
   * @param <T> The type of the desired Lazy object
   * @param v The value to be used
   * @return A Lazy object with the given value
   */
  public static <T> Lazy<T> of(T v) {
    return new Lazy<T>(Maybe.some(v), null);
  }

  /**
   * Initializes the Lazy object with the given producer.
   *
   * @param <T> The type of the desired Lazy object
   * @param s The producer that produces the value when needed
   * @return A Lazy object with the given value
   */
  public static <T> Lazy<T> of(Producer<? extends T> s) {
    return new Lazy<T>(Maybe.none(), s);
  }

  /**
   * Called when the value is needed.
   * The computation should only be done once for the same value.
   *
   * @return If the value is already available, return that value;
   otherwise, compute the value and return it.
   */
  public T get() {
    T v = this.value.orElseGet(this.producer);
    this.value = Maybe.some(v);
    return v;
  }

  /**
   * Lazily maps the value of the instance.
   *
   * @param <U> The type of the mapped Lazy object
   * @param transformer The given transformer
   * @return A new Lazy instance with the value inside it transformed.
   The transformer is only evaluated once.
   */
  public <U> Lazy<U> map(Transformer<? super T, ? extends U> transformer) {
    return Lazy.of(() -> transformer.transform(this.get()));
  }

  /**
   * Lazily maps the value of the instance.
   * Similar to map, but prevents nested Lazy instances.
   *
   * @param <U> The type of the mapped Lazy object
   * @param transformer The given transformer
   * @return A new `Lazy` instance with the value inside it transformed.
   The transformer is only evaluated once.
   */
  public <U> Lazy<U> flatMap(Transformer<? super T,
      ? extends Lazy<? extends U>> transformer) {
    return Lazy.of(() -> transformer.transform(this.get()).get());
  }

  /**
   * Lazily tests if the value passes the test or not.
   *
   * @param cond The condition to test the value with
   * @return A Lazy instance that reflects the result of the test
   */
  public Lazy<Boolean> filter(BooleanCondition<? super T> cond) {
    return Lazy.of(() -> cond.test(this.get()));
  }

  /**
   * Lazily combines the values of two Lazy instances.
   *
   * @param <S> The type of the second Lazy object
   * @param <R> The type of the combined Lazy object
   * @param other The other Lazy instance to be combined with
   * @param combiner The combiner to be used
   * @return A new Lazy instance that contains the combined result
   */
  public <S, R> Lazy<R> combine(Lazy<? extends S> other,
      Combiner<? super T, ? super S, ? extends R> combiner) {
    return Lazy.of(() -> combiner.combine(this.get(), other.get()));
  }

  /**
   * Return the string representation of the list.
   *
   * @return The string representation of the list.
   */
  @Override
  public String toString() {
    return this.value.map(t -> String.valueOf(t)).orElse("?");
  }

  /**
   * Checks the semantic equality with another object.
   *
   * @param o The object to be compared with
   * @return true only both objects being compared are Lazy
   and the value contains within are equals (according to their equals() meth
   */
  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null || !(o instanceof Lazy<?>)) {
      return false;
    }

    Lazy<?> other = (Lazy<?>) o;

    // semantic comparison here
    return other.get().equals(this.get());
  }
}
```