Generic Java

Rushi Shah

12 September 2016

So recently for my Data Structures class, I implemented a Trie in Java that was used to represent information for a simple Markov chain application. I made the code as polymorphic as possible using a Java feature called generics (the things that let you make an ArrayList that contains any type). This meant that I often wrote code that looked a little something like this: `ArrayList<TNode<Freq<Character>>>`

I personally think that is one too many sets of angle brackets for basic Java code, but at least it worked pretty polymorphically.

1 Background

The assignment was to create a weak artificial intelligence that mimics a corpus of text using the probability distributions of character frequencies.

1.1 Markov Chains

This is basically a Markov chain over a text file that examines the probability that any character will follow a sequence of seed characters. The way the assignment explained it was:

> "A (discrete) Markov Chain is a sequence where each value depends only upon the previous value (or k previous values)"

Each letter of the resulting text is just dependent on the previous value (or k previous values).

1.2 Tries

In order to keep track of the number of times each character followed any other prefix of characters, I used a structure called a "trie". Trie's are used to represent data structures that define a path to a value. In this case, the path is the seed provided and the value is the next character. The trie used in this project had a depth of k + 1, where k represents the length of the prefix and where the leaves represented what characters follow the prefix of the path used to get to them.

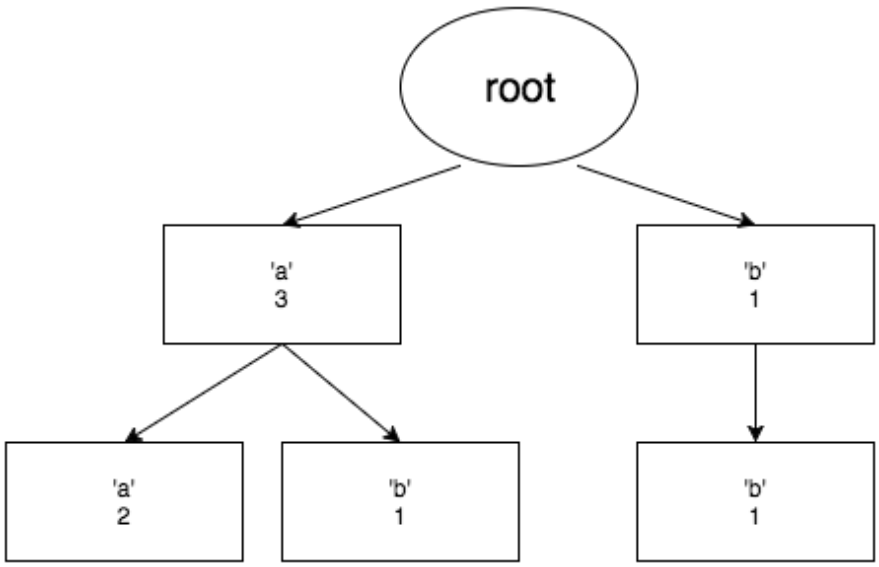

Figure 1: Trie for the string "aaabb" with a k value of 1.

Operations on trie's (inserting children, etc.) are not dependent on the type of the object the trie contains. Thus, polymorphism was in order, and the implementation of this polymorphism fell to generics. Since it was my first time getting extremely comfortable with generics in Java, a short write up follows:

2 Generics

Utilizing generics in Java was actually not as awful of an experience as I had expected. They were surprisingly useful, and I pretty much just thought of them in relation to kinds in Haskell.

2.1 Freq<O>

Each node in the trie needed to keep track of the value it represented and the frequency that value appeared in that location. Thus, a class `Freq` would have two attributes (`cargo` and `frequency`). However, the type of `cargo` depends on what type is being used at the time (Character? String? Integer?). Thus, the type of cargo is abstracted away with the generic O.

For example, assume you had a type called `FreqChar` that contained cargo::Character. After you were done defining that class, you could go through and replace every "Character" with "O" and things should work more or less the same.

Equality testing also needed to be supported, but overriding the `equals` method presented a problem. In order to do so, you would need to compare cargo, and you would need to check if the Object passed in is an instance of Freq with the same type as cargo. The following code goes around this difficulty by casting the object as the correct type (question-marks are used because the specific type doesn't really matter):

```java
@Override
public boolean equals(Object o) {
    if (o instanceof Freq<?>) {
        Freq<?> oc = ((Freq<?>) o);
        return oc.getCargo().equals(cargo);
    } else {
        return false;
    }
}
```

2.2 TNode<T>

Each node of the trie also needed to be polymorphic because certain operations on a trie (such as inserting a series of nodes) is defined similarly regardless of what type of values the trie holds. Thus, this type can be abstracted away with `TNode<T>` where T represents the type of object being held in the trie.

Note that the trie is implemented as a series of interlinked nodes rather than the structure being defined as a whole. This means that each trie has a series of sub-tries attached to it recursively.

Thus, functions like insertChild are also defined recursively. Consider inserting a series of children where the first piece of what you're trying to insert is already in the trie. In order to do so, you must find that object, increment it's frequency, and move down to it where you can insert the remainder of what you need to insert. Assume, for example, you need to insert a series of elements in which the head is `a` and the tail is `arr`. You know the TNode will hold a Freq object (which defines an increment method), but you need to encode that in the insertChild function as follows:

```java
//...
int n = children.indexOf(a);
if (n >= 0) {
    T c = children.get(n).getValue();
    if (c instanceof Freq<?>) {
        ((Freq<?>) c).inc();
    }
    children.get(n).insertChildren(arr);
}
//...
```

On second thought, I am sure there is a way to ensure that the type parameter T is-a Freq<O> in the definition of TNode. I guess doing so is left as an exercise for the reader...

The same pattern for checking equality is used in TNode as was used in Freq.

3 Conclusion

Okay, in this case, polymorphism in Java and the type system weren't too bad. They didn't get in the way as often as I had remembered Java always seemed to. With that being said, I think the `instanceof` calls scared me, they seem to destroy the idea of type-safety. Also, I was annoyed it isn't immediately obvious how to enforce parameters on the types that can be contained. In Haskell you can do something like

```
class (Integral a) => MyEq a where
  e :: a -> a -> Bool
```

which ensures that the only type parameters that can be passed into `MyEq` must be integer values. All in all, generics in Java are nice, but writing them just gave me a deeper appreciation of Haskell.