Simulating Higher Kinded Types in Java

This article introduces the concept of Higher Kinded Types for Java developers. Java does not natively support Higher Kinded Types, so we will take a deep dive into an innovative approach pioneered by Daniel Gronau and team behind the advanced functional library HighJ.

This article forms part of the series on DSLs with the Free Monad in Java 8, and provides a bridge for those moving from part I (which relies on standard Java generics) to the forthcoming part II (which makes use of simulated higher kinded types).

Intro to Higher Kinded Types

There is more than one type of polymorphism

Most Java developers are familiar with inheritance (or sub-type polymorphism), and likewise most Java developers are familiar with generics. They may be less aware that leveraging generics within your code to make it more reusable is itself a form of polymorphism called parametric polymorphism.

Towards a Functor Typeclass

In Part I of, DSLs with the Free Monad in Java 8, we made use of the Transformable interface from cyclops-react to implement sub-types that can be transformed via a map method. Transformables are Functors which can be implemented using traditional sub-type polymorphism (inheritance) as follows.

public interface Functor<T>{

<R> Functor<R> map(Function<? super T, ? extends R> mapper);

}

This forces the developer to implement Functor (or Transformable) on all of their DSLs commands, introducing a tight coupling between the command definition and it’s Functor implementation. Parametric polymorphism (aka generics) could allow us to decouple these concerns, keeping the implementation of the map method separate from the definition of the entities that make up our DSL’s commands. Unfortunately Java’s Generics aren’t quite powerful enough to do this natively.

Take a look at the code below

public interface Functor<C> {

public <T,R> C<R> map(Function<? super T, ? extends R> mapper, C<T> genericType);

}

At first blush this definition of a Functor interface allows us to implement map in a generic form separately from the type being transfomed (e.g. in the context of Part I these are our DSL Command objects). We could define separate Functor implementations for each Command type C, and use composition rather than inheritance in our code (by passing both the Functor implementation and Command into Free, for example). The command interface could disappear, and indeed our commands themselves would become standard POJOs e.g.

public class Output<T>{

private final char c;

private final T next;

private Output(final char c, final T next) {

this.c = c;

this.next = next;

}

}

public class OutputFunctor implements Functor<Output>{

@Override

public <R> Output<R> map(final Function<? super T, ? extends R> f,Output<T> output) {

return new Output<>(c, f.apply(next));

}

}

Unfortunately, however, this code will not compile!

In Java we can’t define a generic type C that takes an additional generic type R. We can’t put generics on generics in Java, at least not directly.

Simulated higher kinded parametricity

All problems in computer science can be solved by another level of indirection — David Wheeler

The solution to this is to define an interface that captures and stores the two generic types we need separately. In our impossible Functor below that means capturing C and T/R.

public interface Functor<C> {

public <T,R> C<R> map(Function<? super T, ? extends R> mapper, C<T> genericType);

}

The core type WITNESS (<C>) e.g. something that represents Output, Bell, Done

The data Type(<T>,<R>) that the core type accepts / stores (Output<DATA\_TYPE>, Bell<DATA\_TYPE> etc)

We can define an interface called Higher to capture the core type WITNESS and the current data type.

public interface Higher<WITNESS,T>{ }

Understanding Higher Kinded Types with a j.u.List

In this section we will create a Higher Kinded encoding of a java.util.List in step-by-step fashion to investigate & understand what the best practices for simulating higher kinded types are. We will create a class that implements both Higher and List, and that uses a static inner class as the Witness Type.

Expressing a j.u.List in higher level generics

To delve a little deeper into how Witness types simulate Higher Kinded Types let’s take a look at how a plain old Java List might look when encoded using this technique.

A naive first approach might be to use the List itself as the Witness type

Higher<List,T>

Higher<List<?>,T>

But because List itself is generic we are forced to either ignore the List’s generic type parameter or replace it with a wildcard. A better approach pioneered by the advanced functional libary HighJ is to use a non-generic proxy (or Witness) type to represent the List.

Higher<ListKind,T>

We can define ListKind as follows, note that it must :

Implement Higher<ListKind,T>

Provide a wrapper around a j.u.List

public class ListKind<T> implements Higher<ListKind,T>{

private List<T> boxed;

public ListKind(List<T> list){

boxed= list;

}

public static <T> List<T> narrowK(Higher<ListKind,T> hkt){

ListKind<T> list = (ListKind<T>)hkt;

return list.boxed;

}

}

We can use the narrowK method to convert between Higher Kinded encodings of Lists and the JDK interface.

List<Integer> list = Arrays.asList(1,2,3);

ListKind<Integer> listKind = new ListKind(list);

Higher<ListKind,Integer> htk = listKind;

List<Integer> decoded = ListKind.narrowK(hkt);

Now we have a way of encoding List’s as Higher instances and decoding them back to j.u.Lists.

Reducing the usability gap between higher type and the core type

One challenge with our implementation of ListKind is that it is not actually a List itself, performing List operations on a ListKind requires a conversion before each List operation, and if a HKT encoded type is needed, a conversion after as well.

We can make ListKind implement List and turn our boxed List instance into a delegate. Once ListKind implements List it becomes generic. Generic types are less suitable for use as Witness types. To circumvent this we can create a static inner class µ for our List.

The name ‘µ’ (pronounced mu) is used by convention for inner classes that are used to indicate what the real concrete type should be. In the example below Higher<ListKind.µ,T> corresponds to an encoded List<T> simply because ListKind<T> implements Higher<ListKind.µ,T> and List<T>

public class ListKind<T> implements List<T>,Higher<ListKind.µ,T>{

public static class µ { }

private List<T> delegate;

public static ListKind<T> of(T... array){

return new ListKind<T>(array);

}

public static <T> List<T> narrowK(Higher<ListKind,T> hkt){

return (ListKind<T>)hkt;

}

@Override

public int size(){

return delegate.size();

}

...

}

Now we can use our ListKind as a List much more naturally.

ListKind<Integer> listKind = ListKind.of(1,2,3);

List<Integer> list = listKind;

Higher<ListKind.µ,Integer> htk = listKind;

listKind.add(10);

Creating a Functor for Lists

Higher<ListKind.µ,Integer> htk;

Using Higher to encode two pieces of generic information separately, we can define a far more abstract version of map method that will actually compile in Java.

public interface Functor<WITNESS> {

<T,R> Higher<WITNESS,R> map(Function<? super T, ? extends R> fn, Higher<WITNESS, T> ds);

}

This is actually pretty close to the definition of the Functor typeclass seen in many functional languages and libraries (and is very similar to the one defined in cyclops-react) . One of the key characteristics of a Functor is that it is transformable by a well behaved implementation of map.

We could define a Functor for Lists as follows

public class ListFunctor implements Functor<ListKind.µ>{

public <T,R> Higher<ListKind.µ,R> map(Function<? super T, ? extends R> fn, Higher<ListKind.µ,T> ds){

return ListKind.narrowK(ds)

.stream()

.map(fn)

.collect(()->ListKind.of());

}

}

And to call it

ListKind<Integer> list = ListKind.of(1,2,3);

ListFunctor functor = new ListFunctor()

List<Integer> doubled = ListKind.narrowK(functor.map(i->i\*2,list));

(Note: there is no need to create your own Higher Kinded encodings and typeclasses for common Java types, cyclops-react defines HKT encodings and Typeclasses for Streams, Optionals, CompletableFutures, Lists and more).

cyclops HKTs and Typeclasses

In cyclops-react many datatypes already implement Higher natively. For example we can define a List and use it in Higher form directly

Higher<ListX.µ,Integer> hkt = ListX.of(1,2,3);

The same is true for a Stream

Higher<ReactiveSeq.µ,Integer> hkt = ReactiveSeq.of(1,2,3);

Typeclasses are predefined for many types and are available via Instances. E.g for a List

Functor<ListX.µ> functor = ListX.Instances.functor();

Higher<ListX.µ, Integer> hkt = functor.map(i -> i \* 2, ListX.of(1, 2, 3));

ListX<Integer> list = hkt.convert(ListX::narrowK);

or an Optional

Functor<OptionalKind.µ> functor = Optionals.Instances.functor();

Higher<OptionalKind.µ, Integer> hkt = functor.map(i -> i \* 2, OptionalKind.widen(Optional.of(3)));

Optional<Integer> opt = OptionalKind.narrowK(hkt);

Appropriate levels of generalization

Simulating higher kinded types in this fashion allows us to generalize across concepts such as ‘things that can be transformed’, or more specificaly Functors, Monads, ApplicativeFunctors from the functional world. Most of the team Java developers do not need to do this. Most of the time, where you need to use a List, you should use a List. If you would like to use a supercharged j.u.List that can act as a functor, monad, applicative functor and more using ListX from cyclops-react directly may be what you need.

As we shall see in the continuing series DSLs with the Free Monad in Java 8, Implementing and using Free, is one of the use cases where this deeper level of abstraction can add power and simplification.

//standard Java Generics

public static <T> Unrestricted<T> liftF(final Transformable<T> functor);

//HKT encoded

public static <F, T> Free<F, T> liftF(final Higher<F, T> value, final Functor<F> functor);