Higher-rank and higher-kinded types

April 23, 2016

If you enjoy statically-typed functional programming, you've probably heard of higher-rank and higher-kinded types; otherwise, perhaps you might like a quick primer? 😊

Background: what is parametric polymorphism?

You may be familiar with "generics" in Java, or "templates" in C++. *Parametric polymorphism* [1] is the jargon that characterizes these features. It allows us to abstract types from values, analogous to how ordinary functions abstract values from values. For example, consider this function which chooses one of two strings at random:

```
String chooseString(String head, String tail) {
  if (Math.random() < 0.5) {
    return head;
  } else {
    return tail;
  }
}
```

We might want to choose between values of other types too, so we abstract `String` away from this method as a generic type parameter `T`:

```
<T> T chooseValue(T head, T tail) {
  if (Math.random() < 0.5) {
    return head;
  } else {
    return tail;
```

```
}
```

Now we can compute `chooseValue("heads", "tails")` or `chooseValue(0, 1)` or apply values of any other type, as long as both arguments have the same type. That's the essence of parametric polymorphism.

Higher-rank types

Parametric polymorphism allows for "functions" from types to values. In the example above, `chooseValue` is a function which (implicitly) takes a type `T`, and then two arguments of type `T`, and returns a `T`.

You may know that in most modern programming languages, functions are first-class values—they can be stored in variables, passed to functions, etc. In most statically-typed programming languages, however, polymorphic functions are not first-class. You cannot pass `chooseValue` to another function or store it in a variable, and then apply it polymorphically later on. Whenever you want to refer to it, you must specify up front (explicitly or implicitly) a type for `T`, thereby converting it into a monomorphic function.

The idea of higher-rank types is to make polymorphic functions first-class, just like regular (monomorphic) functions.[2] Here's a contrived example that requires higher-rank types, and is thus not valid Java:

```
String chooseStringWeird(
  <T> BiFunction<T, T, T> randomChoice,
  String head,
  String tail
) {
  if (randomChoice(true, false)) {
    return randomChoice(head, tail);
  } else {
    return randomChoice(tail, head);
  }
}

String bestEditor() {
  return chooseStringWeird(chooseValue, "emacs", "vim");
```

```
}
```

Higher-rank types don't come up very often, and supporting them has the unfortunate consequence of making complete type inference undecidable.[3] Even Haskell, the proverbial playground for popular polymorphism paradigms, requires a language extension to support them.

Higher-kinded types

If parametric polymorphism involves functions from types to values, you might wonder about functions from types to types. These type-level functions are called *type operators*. An example of a type operator in Java is `Stack`:

```
Stack<String> names;
```

`Stack` can be thought of as a function that takes a type (in this case, `String`) and returns a type. With higher-kinded types, type operators are first-class types, similar to how higher-rank types treat polymorphic functions as first class values. In other words, type operators can be parameterized by other type operators. Java doesn't support higher-kinded types, but, if it did, you would be able to write something like this:

```
class GraphSearch<T> {
  T<Node> frontier;
}
```

`GraphSearch` is a type operator, and its parameter `T` stands for another type operator such as `Stack` or `Queue`:

```
GraphSearch<Stack> depthFirstSearch;
GraphSearch<Queue> breadthFirstSearch;
```

The "type" of a type is called a *kind*. The kind of an ordinary ("proper") type is usually written `*`, and type constructors (unary type operators) have kind `* -> *`. `GraphSearch` has kind

`(* -> *) -> *`. Types such as `GraphSearch` are called *higher-kinded* because they are higher-order functions on the level of types, and they are "typed" using kinds.

Unlike higher-rank types, higher-kinded types are commonplace in typed functional programming. `Functor` and `Monad` are examples of higher-kinded polymorphism in Haskell. `Monad` in particular is a simple idea, but has a reputation for being difficult to understand for non-functional programmers. I suspect part of this confusion is due to the fact that `Monad` is higher-kinded, which is an idea most programmers are not accustomed to. Maybe this article will help! 😊

Conclusion

In summary, parametric polymorphism introduces functions from types to values, and type operators are functions from types to types. When these functions are "first-class" in the sense that they can be used in a higher-order fashion, they are called *higher-rank* and *higher-kinded*, respectively.

There is one possibility we haven't explored: functions from values to types. These are called "dependent types" and open up a mind-blowing world of programming with proofs. Dependent types are outside the scope of this article, but they are 100% worth learning about if you are interested in type theory.

Speaking of further learning, I highly recommend Benjamin Pierce's book *Types and Programming Languages*, which is, in my opinion, lightyears ahead of any other introductory resource on type systems. Chapter 23 introduces parametric polymorphism with higher-rank types (System F), and Chapter 29 discusses higher-kinded types (System λ_ω).

I hope that was helpful! Let me know in the comments if anything could be explained better.

Footnotes

[1] The term "polymorphism" is autological. In addition to parametric polymorphism, it can also refer to subtype polymorphism or ad hoc polymorphism.
[2] This kind of polymorphism is also called *rank-n polymorphism* or *first-class polymorphism*.
[3] This negative result is from J.B. Wells, 1999.

Does Google execute JavaScript?

Formalizing dynamic scoping

Matthias • 2 years ago

Well written article--thanks!

While Java does not have first-class support for higher-kinded types, are there ways to approximate them? Here's a Kotlin project that attempts to do that, and it looks like it can be translated basically 1:1 to Java:
https://github.com/kotlinz/...

Kotlinz provides a number of interfaces that represent kinds with varying arity by "flattening" the type parameters. For example, F[A, B] becomes K2[F, A, B]. Here's how Functor is implemented:
https://github.com/kotlinz/...

i.e. T can be a type constructor. I think the main problem with these approaches is that Java/Kotlin are not able to express type classes, so existing library types such as List need to be decorated manually first to conform to these interfaces, whereas in Scala this can happen behind the scenes via implicits.

1 • Reply • Share ›

Lii ➚ Matthias • 2 years ago

The difference between a higher-kinded type system and these techniques in Kotlin is that they rely in part on dynamic type checks.

This
can be seen in the implementation of `MaybeFunctor` which uses the `Maybe.narrow` method. `narrow` performs a downcast at runtime from `K1` to `Maybe`.

https://github.com/kotlinz/...

The
class `Maybe.T` is meant to be used only for `K1`s that are implemented by `Maybe`, but there is not way for the compiler to verify that, it
relies on the programmer to do so. Someone technically could use this type for another `K1` implementation, call `MaybeFunctor.fmap` with it, and the cast would fail.

It seem like the result in Kotlin is
pretty type safe, to the point where you have to try hard to get a