Если вы видите что-то необычное, просто сообщите мне.

Haskell Typeclasses as Inheritance

Welcome to part four of our series comparing Haskell's data types to other languages. As I've expressed before, the type system is one of the key reasons I enjoy programming in Haskell. And in this part, we're going to get to the heart of the matter. We'll compare Haskell's typeclass system with the idea of inheritance used by object oriented languages. We'll close out the series in part 5 by talking about type families!

If Haskell's simplicity inspires you as well, try it out! Download our Beginners Checklist and read our Liftoff Series to get going!

You can also studies these code examples on your own by taking a look at our Github Repository! For this part, here are the respective files for the Haskell, Java and Python examples.

#TYPECLASSES REVIEW Before we get started, let's do a quick review of the concepts we're discussing. First, let's remember how typeclasses work. A typeclass describes a behavior we expect. Different types can choose to implement this behavior by creating an instance.

One of the most common classes is the Functor typeclass. The behavior of a functor is that it contains some data, and we can map a type transformation over that data.

In the raw code definition, a typeclass is a series of function names with type signatures. There's only one function for Functor: fmap:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

A lot of different container types implement this typeclass. For example, lists implement it with the basic map function:

```
instance Functor [] where
  fmap = map
```

But now we can write a function that assumes nothing about one of its inputs except that it is a functor:

```
stringify :: (Functor f) => f Int -> f String
```

We could pass a list of ints, an IO action returning an Int, or a Maybe Int if we wanted. This function would still work! This is the core idea of how we can get polymorphic code in Haskell.

INHERITANCE BASICS

As we saw in previous parts, object oriented languages like Java, C++, and Python tend to use inheritance to achieve polymorphism. With inheritance, we make a new class that extends the functionality of a parent class. The child class can access the fields and functions of the parent. We can call functions from the parent class on the child object. Here's an example:

```
public class Person {
  public String firstName;
  public int age;

  public Person(String fn, int age) {
    this.firstName = fn;
    this.age = age;
  }

  public String getFullName() {
    return this.firstName;
  }
}

public class Employee extends Person {
  public String lastName;
  public String company;
  public String email;
  public int salary;
```

```
  public Employee(String fn,
                  String ln,
                  int age,
                  String company,
                  String em,
                  int sal) {
    super(fn, age);
    this.lastName = ln;
    this.company = company;
    this.email = em;
    this.salary = sal;
  }

  public String getFullName() {
    return this.firstName + " " + this.lastName;
  }
}
```

Inheritance expresses an "Is-A" relationship. An Employee "is a" Person. Because of this, we can create an Employee, but pass it to any function that expects a Person or store it in any data structure that contains Person objects. We can also call the getFullName function from Person on our Employee type, and it will use the Employee version!

```
public static void main(String[] args) {
  Employee e = new Employee("Michael", "Smith", 23, "Google", "msmith@google.com", 100000);
  Person p = new Person("Katie", 25);
  Person[] people = {e, p};
  for (Person person : people) {
    System.out.println(person.getFullName());
  }
}
```

This provides a useful kind of polymorphism we can't get in Haskell, where we can't put objects of different types in the same list.

BENEFITS

Inheritance does have a few benefits. It allows us to reuse code. The Employee class can use the getFullName function without having to define it. If we wanted, we could override the definition in the Employee class, but we don't have to.

Inheritance also allows a degree of polymorphism, as we saw in the code examples above. If the circumstances only require us to use a Person, we can use an Employee or any other subclass of Person we make.

We can also use inheritance to hide variables away when they aren't needed by subclasses. In our example above, we made all our instance variables public. This means an Employee function can still call this.firstName. But if we make them private instead, the subclasses can't use them in their functions. This helps to encapsulate our code.

DRAWBACKS

Inheritance is not without its downsides though. One unpleasant consequence is that it creates a tight coupling between classes. If we change the parent class, we run the risk of breaking all child classes. If the interface to the parent class changes, we'll have to change any subclass that overrides the function.

Another potential issue is that your interface could deform to accommodate child classes. There might be some parameters only a certain child class needs, and some only the parent needs. But you'll end up having all parameters in all versions because the API needs to match.

A final problem comes from trying to understand source code. There's a yo-yo effect that can happen when you need to hunt down what function definition your code is using. For example your child class can call a parent function. That parent function might call another function in its interface. But if the child has overridden it, you'd have to go back to the child. And this pattern can continue, making it difficult to keep track of what's happening. It gets even worse the more levels of a hierarchy you have.

I was a mobile developer for a couple years, using Java and Objective C. These kinds of flaws were part of what turned me off OO-focused languages.

TYPECLASSES AS INHERITANCE Now, Haskell doesn't allow you to "subclass" a type. But we can still get some of the same effects of inheritance by using typeclasses. Let's see how this works with the Person example from above. Instead of making a separate Person data type, we can make a Person typeclass. Here's one approach:

```
class Person a where
  firstName :: a -> String
  age :: a -> Int
  getFullName :: a -> String

data Employee = Employee
  { employeeFirstName :: String
  , employeeLastName :: String
  , employeeAge :: Int
  , company :: String
  , email :: String
  , salary :: Int
  }

instance Person Employee where
  firstName = employeeFirstName
  age = employeeAge
  getFullName e = employeeFirstName e ++ " " ++ employeeLastName e
```

We can one interesting observation here. Multiple inheritance is now trivial. After all, a type can implement as many typeclasses as it wants. Python and C++ allows multiple inheritance. But it presents enough conceptual pains that languages like Java and Objective C do not allow it.

Looking at this example though, we can see a big drawback. We won't get much code reusability out of this. Every new type will have to define getFullName. That will get tedious. A different approach could be to only have the data fields in the interface. Then we could have a library function as a default implementation:

```
class Person a where
  firstName :: a -> String
  lastName :: a -> String
  age :: a -> Int
```

```
getFullName :: (Person a) => a -> String
getFullName p = firstName p ++ " " ++ lastName p

data Employee = ... (as above)

instance Person Employee where
  firstName = employeeFirstName
  age = employeeAge
  -- getFullName defined at the class level.
```

This allows code reuse. But it does not allow overriding, which the first example would. So you'd have to choose on a one-off basis which approach made more sense for your type. And no matter what, we can't place different types into the same array, as we could in Java.

While Java inheritance stresses the importance of the "Is-A" relationship, typeclasses are more flexible. They can encode "Is-A", in the way the "A List is a Functor". But oftentimes it makes more sense to think of them like Java interfaces. When we think of the Eq typeclass, it tells us about a particular behavior. For example, a String is equatable; there is an action we can take on it that we know about. Or it can express the "Has-A" relationship. In the example above, rather than calling our class Person, we might just limit it to HasFullName, with getFullName being the only function. Then we know an Employee "has" a full name.

TRYING AT MORE DIRECT INHERITANCE

If you've spent any time in a debugger with Java or Objective C, you quickly pickup on how inheritance is actually implemented under the hood. The "child" class actually has a pointer to the "parent" class instance, so that it can reference all the shared items. We can also try to mimic this approach in Haskell as well:

```
data Person2 = Person2
  { firstName' :: String
  , age' :: Int
  }
```

```
data Employee2 = Employee2
  { employeePerson :: Person2
  , company' :: String
  , email' :: String
  , salary' :: Int
  }
```

Now whenever we wanted to access the person, we could use the employeePerson field and call Person functions on it. This is a reasonable pattern in certain circumstances. It does allow for code re-use. But it doesn't allow for polymorphism by itself. We can't automatically pass an Employee to functions that demand a Person. We must either un-wrap the Person each time or wrap both data types in a class. This pattern gets more unsustainable as you add more layers to the inheritance (e.g. having Engineer inherit from Employee.

JAVA INTERFACES

Now it's important to note that Java does have another feature that's arguably more comparable to typeclasses, and this is the Interface. An interface specfies a series of actions and behavior. So rather than expressing a "Is-A" relationship, an interface express a "Does" relationship (class A "does" interface B). Let's explore a quick example:

```
public interface PersonInterface {
  String getFullName();
}

public class Adult implements PersonInterface {
  public String firstName;
  public String lastName;
  public int age;
  public String occupation;

  public Adult(String fn, String ln, int age, String occ) {
    this.firstName = fn;
    this.lastName = ln;
    this.age = age;
```

```
    this.occupation = occ;
  }

  public String getFullName() {
     return this.firstName + " " + this.lastName;
  }
}
```

All the interface specifies is one or more function signatures (though it can also optionally define constants). Classes can choose to "implement" an interface, and it is then up to the class to provide a definition that matches. As with subclasses, we can use interfaces to provide polymorphic code. We can write a function or a data structure that contains elements of different types implementing PersonInterface, as long as we limit our code to calling functions from the interface.

PYTHON INHERITANCE AND INTERFACES

Back in part 3 we explored basic inheritance in Python as well. Most of the ideas with Java apply, but Python has fewer restrictions. Interfaces and "behaviors" often end up being a lot more informal in Python. While it's possible to make more solid contracts, you have to go a bit out of your way and explore some more advanced Python features involving decorators.

We have some simple Python examples here but the details aren't super interesting on top of what we've already looked at.

COMPARISONS

Object oriented inheritance has some interesting uses. But at the end of the day, I found the warts very annoying. Tight coupling between classes seems to defeat the purpose of abstraction. Meanwhile, restrictions like single inheritance feel like a code smell to me. The existence of that restriction suggests a design flaw. Finally, the issue of figuring out which version of a function

you're using can be quite tricky. This is especially true when your class hierarchy is large.

Typeclasses express behaviors. And as long as our types implement those behaviors, we get access to a lot of useful code. It can be a little tedious to flesh out a new instance of a class for every type you make. But there are all kinds of ways to derive instances, and this can reduce the burden. I find typeclasses a great deal more intuitive and less restrictive. Whenever I see a requirement expressed through a typeclass, it feels clean and not clunky. This distinction is one of the big reasons I prefer Haskell over other languages.

CONCLUSION

That wraps up our comparison of typeclasses and inheritance! There's one more topic I'd like to cover in this series. It goes a bit beyond the "simplicity" of Haskell into some deeper ideas. We've seen concepts like parametric types and typeclasses. These force us to fill in "holes" in a type's definition. We can expand on this idea by looking at type families in the fifth and final part of this series!

If you want to stay up to date with our blog, make sure to subscribe! That will give you access to our subscriber only resources page!

Revision #1
Created 11 March 2022 06:06:48 by gasick
Updated 11 March 2022 17:11:17 by gasick