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

Parameterized Types in Haskell

Welcome back to our series on the simplicity of Haskell's data declarations. In part 2, we looked at how to express sum types in different languages. We saw that they fit very well within Haskell's data declaration system. For Java and Python, we ended up using inheritance, which presents some interesting benefits and drawbacks. We'll explore those more in part 4. But first, we should wrap our heads around one more concept: parametric types.

We'll see how each of these languages allows for the concept of parametric types. In my view, Haskell does have the cleanest syntax. But other compiled languages do pretty well to incorporate the concept. Dynamic languages though, provide insufficient guarantees for my liking.

This all might seem a little wild if you haven't done any Haskell at all yet! Read our Liftoff Series to get started!

As always, you can look at the code for these articles on our Github Repository! For this article you can look at the Haskell example, or the Java code, or the Python example.

HASKELL PARAMETRIC TYPES Let's remember how easy it is to do parametric types in Haskell. When we want to parameterize a type, we'll add a type variable after its name in the definition. Then we can use this variable as we would any other type. Remember our Person type from the first part? Here's what it looks like if we parameterize the occupation field.

```
data Person o = Person
  { personFirstName :: String
  , personLastName :: String
  , personEmail :: String
  , personAge :: Int
  , personOccupation :: o
  }
```

We add the o at the start, and then we can use o in place of our former String type. Now whenever we use the Person type, we have to specify a type parameter to complete the definition.

```
data Occupation = Lawyer | Doctor | Engineer

person1 :: Person String
person1 = Person "Michael" "Smith" "msmith@gmail.com" 27 "Lawyer"

person2 :: Person Occupation
person2 = Person "Katie" "Johnson" "kjohnson@gmail.com" 26 Doctor
```

When we define functions, we can use a specific version of our parameterized type if we want to constrain it. We can also use a generic type if it doesn't matter.

```
salesMessage :: Person Occupation -> String
salesMessage p = case personOccupation p of
  Lawyer -> "We'll get you the settlement you deserve"
  Doctor -> "We'll get you the care you need"
  Engineer -> "We'll build that app for you"

fullName :: Person o -> String
fullName p = personFirstName p ++ " " ++ personLastName p
```

Last of all, we can use a typeclass constraint on the parametric type if we only need certain behaviors:

```
sortOnOcc :: (Ord o) => [Person o] -> [Person o]
sortOnOcc = sortBy (\p1 p2 -> compare (personOccupation p1) (personOccupation p2))
```

JAVA GENERIC TYPES

Java has a comparable concept called generics. The syntax for defining generic types is pretty clean. We define a type variable in brackets. Then we can use that variable as a type freely throughout the class definition.

```
public class Person<T> {
    private String firstName;
    private String lastName;
    private String email;
    private int age;
    private T occupation;

    public Person(String fn, String ln, String em, int age, T occ) {
        this.firstName = fn;
        this.lastName = ln;
        this.email = em;
        this.age = age;
        this.occupation = occ;
    }

    public T getOccupation() { return this.occupation; }
    public void setOccupation(T occ) { this.occupation = occ; }
    ...
}

enum Occupation {
  LAWYER,
  DOCTOR,
  ENGINEER
}

public static void main(String[] args) {
  Person<String> person1 = new Person<String>("Michael", "Smith", "msmith@gmail.com", 27, "Lawyer");
  Person<Occupation> person2 = new Person<Occupation>("Katie", "Johnson", "kjohnson@gmail.com", 26,
Occupation.DOCTOR);
}
```

There's a bit of a wart in how we pass constraints. This comes from the Java distinction of interfaces from classes. Normally, when you define a class and state the subclass, you would use the extends keyword. But when your class uses an interface, you use the implements keyword.

But with generic type constraints, you only use extends. You can chain constraints together with &. But if one of the constraints is a subclass, it must come first.

public class Person<T extends Number & Comparable & Serializable> { In this example, our template type T must be a subclass of Number. It must then implement the Comparable and Serializable interfaces. If we mix the order up and put an interface before the parent class, it will not compile:

public class Person<T extends Comparable & Number & Serializable> { C++ TEMPLATES For the first time in this series, we'll reference a little bit of C++ code. C++ has the idea of "template types" which are very much like Java's generics. Here's how we can create our user type as a template:

```
template <class T>
class Person {
public:
  string firstName;
  string lastName;
  string email;
  int age;
  T occupation;

  bool compareOccupation(const T& other);
};
```

There's a bit more overhead with C++ though. C++ function implementations are typically defined outside the class definition. Because of this, you need an extra leading line for each of these stating that T is a template. This can get a bit tedious.

```
template <class T>
bool Person::compareOccupation(const T& other) {
  ...
}
```

One more thing I'll note from my experience with C++ templates. The error messages from template types can be verbose and difficult to parse. For example, you could forget the template line above. This alone could cause a very confusing message. So there's definitely a learning curve. I've always found Haskell's error messages easier to deal with.

PYTHON - THE WILD WEST!

Since Python isn't compiled, there aren't type constraints when you construct an object. Thus, there is no need for type parameters. You can pass whatever object you want to a constructor. Take this example with our user and occupation:

```
class Person(object):

  # This definition hasn't changed!
  def __init__(self, fn, ln, em, age, occ):
    self.firstName = fn
    self.lastName = ln
    self.email = em
    self.age = age
    self.occupation = occ

stringOcc = "Lawyer"
person1 = Person(
    "Michael",
    "Smith",
    "msmith@gmail.com",
    27,
    stringOcc)

class Occupation(object):
  def __init__(self, name, location):
    self.name = name
    self.location = location

classOcc = Occupation("Software Engineer", "San Francisco")

# Still works!
person2 = Person(
  "Katie",
  "Johnson",
  "kjohnson@gmail.com",
  26,
```

```
  classOcc)
```

Of course, with this flexibility comes great danger. If you expect there are different types you might pass for the occupation, your code must handle them all! Without compilation, it can be tricky to know you can do this. Someone might see an instance of a "String" occupation and think they can call string functions on it. But these functions won't work for other types!

```
people = [person1, person2]
for p in people:
  # This works. Both types of occupations are printable.
  # (Even if the Occupation output is unhelpful)
  print(p.occupation)

  # This won't work! Our "Occupation" class
  # doesn't work with "len"
  print(len(p.occupation))
```

So while you can do polymorphic code in Python, you're more limited. You shouldn't get too carried away, because it is more likely to blow up in your face.

CONCLUSION

Now that we know about parametric types, we have more intuition for the idea of filling in type holes. This will come in handy for part 4 as we look at Haskell's typeclass system for sharing behaviors. We'll compare the object oriented notion of inheritance and Haskell's typeclasses. This distinction gets to the core of why I've come to prefer Haskell as a language. You won't want to miss it!

If these comparisons have intrigued you, you should give Haskell a try! Download our Beginners Checklist to get started!

Revision #1
Created 11 March 2022 06:04:59 by gasick
Updated 11 March 2022 17:11:17 by gasick