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

Sum Types in Haskell

Welcome to the second part of our series on Haskell's data types. This is part of an exploration of concepts that are simple to express in Haskell but harder in other languages. In part 1 we began by looking at simple data declarations. In this part, we'll go one step further and look at sum types. That is, we'll consider types with more than one constructor. These allow the same type to represent different kinds of data. They're invaluable in capturing many concepts. In you already know about sum types, you should move onto part 3, where we'll get into parameterized types.

Most of the material in this article is pretty basic. But if you haven't gotten the chance to use Haskell yet, you might to start from the beginning! Download our Beginners Checklist or read our Liftoff Series!

Don't forget you can a look at the code for these articles on our Github Repository! You can look along for reference and try to make some changes as well. For this article you can look at the Haskell code here, or the Java examples), or the Python example.

HASKELL BASIC SUM TYPES

In part 1, we started with a basic Person type like so:

data Person = Person String String String Int String We can expand this type by adding more constructors to it. Let's imagine our first constructor refers to an adult person. Then we could make a second constructor for a Child. It will have different information attached. For instance, we only care about their first name, age, and what grade they're in:

```
data Person =
  Adult String String String Int String |
  Child String Int Int
```

To determine what kind of Person we're dealing with, it's a simple case of pattern matching. So whenever we need to branch, we do this pattern match in a function definition or a case statement!

```
personAge :: Person -> Int
personAge (Adult _ _ _ a _) = a
personAge (Child _ a _) = a

-- OR

personAge :: Person -> Int
personAge p = case p of
  Adult _ _ _ a _ -> a
  Child _ a _ -> a
```

On the whole, our definition is very simple! And the approach scales. Adding a third or fourth constructor is just as simple! This extensibility is super attractive when designing types. The ease of this concept was a key point in convincing me about Haskell.

RECORD SYNTAX

Before we move onto other languages, it's worth noting the imperfections with this design. In our type above, it can be a bit confusing what each field represents. We used record syntax in the previous part to ease this pain. We can apply that again on this sum type:

```
data Person2 =
  Adult2
    { adultFirstName :: String
    , adultLastName :: String
    , adultEmail :: String
    , adultAge :: Int
    , adultOccupation :: String
    } |
  Child2
    { childFirstName :: String
    , childAge :: Int
    , childGrade :: Int
```

```
    }
```

This works all right, but it still leaves us with some code smells we don't want in Haskell. In particular, record syntax derives functions for us. Here are a few type signatures of those functions:

```
adultEmail :: Person -> String
childAge :: Person -> Int
childGrade :: Person -> Int
```

Unfortunately, these are partial functions. They are only defined for Person2 elements of the proper constructor. If we call adultEmail on a Child, we'll get an error, and we don't like that. The types appear to match up, but it will crash our program! We can work around this a little by merging field names like adultAge and childAge. But at the end of the day we'll still have some differences in what data we need.

```
-- These compile, but will fail at runtime!
adult2Error :: String
adult2Error = childFirstName adult2

child2Error :: String
child2Error = adultLastName child2
```

Coding practices can reduce the burden somewhat. For example, it is quite safe to call head on a list if you've already pattern matched that it is non-empty. Likewise, we can use record syntax functions if we're in a "post-pattern-match" situation. But we would need to ignore them otherwise! And this is a rule we would like to avoid in Haskell.

JAVA APPROACH I: MULTIPLE CONSTRUCTORS

Now let's try to replicate the idea of sum types in other languages. It's a little tricky. Here's a first approach we can do in Java. We could set a flag on our type indicating whether it's a Parent or a Child. Then we'll have all the different fields within our type. Note we'll use public fields without getters and setters for the sake of simplicity. Like Haskell, Java allows us to use two different

constructors for our type:

```
public class MultiPerson {
  public boolean isAdult;
  public String adultFirstName;
  public String adultLastName;
  public String adultEmail;
  public int adultAge;
  public String adultOccupation;
  public String childFirstName;
  public int childAge;
  public int childGrade;

  // Adult Constructor
  public MultiPerson(String fn, String ln, String em, int age, String occ) {
    this.isAdult = true;
    this.adultFirstName = fn;
    ...
  }

  // Child Constructor
  public MultiPerson(String fn, int age, int grade) {
    this.isAdult = false;
    this.childFirstName = fn;
    ...
  }
}
```

We can see that there's a big amount of bloat on the field values, even if we were to combine common ones like age. Then we'll have more awkwardness when writing functions that have to pattern match. Each function within the type will involve a check on the boolean flag. And these checks might also percolate to outer calls as well.

```
public class MultiPerson {
  ...
  public String getFullName() {
    if (this.isAdult) {
      // Adult Code
    } else {
```

```
      // Child Code
    }
  }
}
```

This approach is harder to scale to more constructors. We would need an enumerated type rather than a boolean for the "flag" value. And it would add more conditions to each of our functions. This approach is cumbersome. It's also very unidiomatic Java code. The more "proper" way involves using inheritance.

JAVA APPROACH II: INHERITANCE

Inheritance is a way of sharing code between types in an object oriented language. For this example, we would make Person a "superclass" of separate Adult and Child classes. We would have separate class declarations for each of them. The Person class would share all the common information. Then the child classes would have code specific to them.

```
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;
  }
}

// NOTICE: extends Person
public class Adult extends Person {
  public String lastName;
```

```
  public String email;
  public String occupation;

  public Adult(String fn, String ln, String em, int age, String occ) {
    // super calls the "Person" constructor
    super(fn, age);
    this.lastName = ln;
    this.email = em;
    this.occupation = occ;
  }

  // Overrides Person definition!
  public String getFullName() {
    return this.firstName + " " + this.lastName;
  }
}

// NOTICE: extends Person
public class Child extends Person {
  public int grade;

  public Child(String fn, int age, int grade) {
    // super calls the "Person" constructor
    super(fn, age);
    this.grade = grade;
  }

  // Does not override getFullName!
}
```

By extending the Person type, each of our subclasses gets access to the firstName and age fields. We also get access to the getFullName function if we want. However, the Adult subclass chooses to override it.

There's a big upside we get here that Haskell doesn't usually have. In this case, we've encoded the constructor we used with the type. We'll be passing around Adult and Child objects for the most part. This saves a lot of the partial function problems we encounter in Haskell.

We will, on occasion, combine these in a form where we need to do pattern matching. For example, we can make an array of Person objects.

Adult adult = new Adult("Michael", "Smith", "msmith@gmail.com", 32, "Lawyer"); Child child = new Child("Kelly", 8, 2); Person[] people = {adult, child}; Then at some point we'll need to determine which have type Adult and which have type Child. This is possible by using the isinstance condition in Java. But again, it's unidiomatic and we should strive to avoid it. Still, inheritance represents a big improvement over our first approach. Luckily, though, we could still use the getFullName function, and it would work properly for both of them with overriding!

PYTHON: ONLY ONE CONSTRUCTOR!

Unlike Java, Python only allows a single constructor for each type. The way we would control what "type" we make is by passing a certain set of arguments. We then provide None default values for the rest. Here's what it might look like.

```
class Person(object):
  def __init__(self,
               fn = None,
               ln = None,
               em = None,
               age = None,
               occ = None,
               grade = None):
    if fn and ln and em and age and occ:
      self.isAdult = true
      self.firstName = fn
      self.lastName = ln
      self.age = age
      self.occupation = occ
      self.grade = None
    elif fn and age and grade:
      self.isAdult = false
```

```
      self.firstName = fn
      self.age = age
      self.grade = grade
      self.lastName = None
      self.email = None
      self.occupation = None
    else:
      raise ValueError("Failed to construct a Person!")

# Note which arguments we use!
adult = Person(fn="Michael", ln="Smith", em="msmith@gmail.com", age=25, occ="Lawyer")
child = Person(fn="Mike", age=12, grade=7)
```

But there's a lot of messiness here! A lot of input combinations lead to errors! Because of this, the inheritance approach we proposed for Java is also the best way to go for Python.

```
class Person():
  def __init__(self, fn, age):
    self.firstName = fn
    self.age = age

  def getFullName(self):
    return self.firstName

class Adult(Person):
  def __init__(self, fn, ln, em, age, occ):
    super().__init__(fn, age)
    self.lastName = ln
    self.email = em
    self.occupation = occ

  def getFullName(self):
    return self.firstName + " " + self.lastName

class Child(Person):
  def __init__(self, fn, age, grade):
    super().__init__(fn, age)
    self.grade = grade
```

Again though, Python lacks pattern matching across different types of classes. This means we'll have more if statements like if isinstance(x, Adult). In fact, these will be even more prevalent in Python, as type information isn't attached.

COMPARISONS

Once again, we see certain themes arising. Haskell has a clean, simple syntax for this concept. It isn't without its difficulties, but it gets the job done if we're careful. Java gives us a couple ways to manage the issue of sum types. One is cumbersome and unidiomatic. The other is more idiomatic, but presents other issues as we'll see later. Then Python gives us a great deal of flexibility but few guarantees about anything's type. The result is that we can get a lot of errors.

CONCLUSION

In this second part of the series,, we continued our look at the simplicity of constructing types in Haskell. We saw how a first try at replicating the concept of sum types in other languages leads to awkward code. In a couple weeks, we'll dig deeper into the concept of inheritance. It offers a decent way to accomplish our task in Java and Python. And yet, there's a reason we don't have it in Haskell. But first up, the next part will look at the idea of parametric types. We'll see again that it is simpler to do this in Haskell's syntax than other languages. We'll need those ideas to help us explore inheritance later.

If this series makes you want to try Haskell more, it's time to get going! Download our Beginner's Checklist for some tips and tools on starting out! Or read our Liftoff Series for a more in depth look at Haskell basics.

Revision #1
Created 2022-03-11 06:02:15 UTC by gasick
Updated 2022-03-11 17:11:17 UTC by gasick