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

Haskell's Data Types!

One of the first major selling points for me and Haskell was the simplicity of data declarations. Even as a novice programmer, I was tired of the odd syntax require just to associate some data together. Haskell was clean and fresh. As I dug deeper, I found that even beyond this, Haskell's approach to data allows some really cool techniques that aren't really present in other languages. In this series, we'll start by learning about Haskell's data syntax, and then we'll explore some of these techniques. Along the way, we'll compare Haskell against some other languages.

- PART 1: HASKELL'S SIMPLE DATA TYPES
- Sum Types in Haskell
- Parameterized Types in Haskell
- Haskell Typeclasses as Inheritance
- Type Families in Haskell

PART 1: HASKELL'S SIMPLE DATA TYPES

I first learned about Haskell in college. I've considered why I kept up with Haskell after, even when I didn't know about its uses in industry. I realized there were a few key elements that drew me to it.

In a word, Haskell is elegant. For me, this means we can simple concepts in simple terms. In this series, we're going to look at some of these concepts. We'll see that Haskell expresses a lot of ideas in simple terms that other languages express in more complicated terms. In the first part, we'll start by looking at simple data declarations. If you're already familiar with Haskell data declarations, you can move onto part 2, where we'll talk about sum types!

If you've never used Haskell, now is the perfect time to start! For a quick start guide, download our Beginners Checklist. For a more in-depth walkthrough, read our Liftoff Series!

Like some of our other beginner-level series, this series has a companion Github Repository. All the code in these articles lives there so you can follow along and try out some changes if you want! For this article you can look at the Haskell definitions here, or the Java code, or the Python example.

HASKELL DATA DECLARATIONS

To start this series, we'll be comparing a data type with a single constructor across a few different languages. In the next part, we'll look at multi-constructor types. So let's examine a simple type declaration:

```
data SimplePerson = SimplePerson String String String Int String
```

Our declaration is very simple, and fits on one line. There's a single constructor with a few different fields attached to it. We know exactly what the types of those fields are, so we can build the object. The only way we can declare a SimplePerson is to provide all the right information in order.

```
firstPerson :: SimplePerson
firstPerson = SimplePerson "Michael" "Smith" "msmith@gmail.com" 32 "Lawyer"
```

If we provide any less information, we won't have a SimplePerson! We can leave off the last argument. But then the resulting type reflects that we still need that field to complete our item:

```
incomplete :: String -> SimplePerson
incomplete = SimplePerson "Michael" "Smith" "msmith@gmail.com" 32

complete :: SimplePerson
complete = incomplete "Firefighter"
```

Now, our type declaration is admittedly confusing. We don't know what each field means at all when looking at it. And it would be easy to mix things up. But we can fix that in Haskell with record syntax, which assigns a name to each field.

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

We can use these names as functions to retrieve the specific fields out of the data item later.

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

With either construction though, we can also use a pattern match to retrieve the relevant information.

```
fullNameSimple :: SimplePerson -> String
fullNameSimple (SimplePerson fn ln _ _ _) = fn ++ " " ++ ln
```

```
fullName' :: Person -> String
fullName' (Person fn ln _ _ _) = fn ++ " " ++ ln
```

And that's the basics of data types in Haskell! Let's take a look at this same type declaration in a couple other languages.

JAVA

If we wanted to express this in the simplest possible Java form, we'd do so like this:

```
public class Person {
  public String firstName;
  public String lastName;
  public String email;
  public int age;
  public String occupation;
}
```

Now, this definition isn't much longer than the Haskell definition. It isn't a very useful definition as written though! We can only initialize it with a default constructor Person(). And then we have to assign all the fields ourselves! So let's fix this with a constructor:

```
public class Person {
    public String firstName;
    public String lastName;
    public String email;
    public int age;
    public String occupation;

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

```
        this.email = em;
        this.age = age;
        this.occupation = occ;
    }
}
```

Now we can initialize it in a sensible way. But this still isn't idiomatic Java. Normally, we would have our instance variables declared as private, not public. Then we would expose the ones we wanted via "getter" and "setter" methods. If we do this for all our types, it would bloat the class quite a bit. In general though, you wouldn't have arbitrary setters for all your fields. Here's our code with getters and one setter.

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

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

  public String getFirstName() { return this.firstName; }
  public String getLastName() { return this.lastName; }
  public String getEmail() { return this.email; }
  public int getAge() { return this.age; }
  public String getOccupation() { return this.occupation; }

  public void setOccupation(String occ) { this.occupation = occ; }
}
```

Now we've got code that is both complete and idiomatic Java.

PUBLIC AND PRIVATE

We can see that the lack of a public/private distinction in Haskell saves us a lot of grief in defining our types. Why don't we do this?

In general, we'll declare our data types so that constructors and fields are all visible. After all, data objects should contain data. And this data is usually only useful if we expose it to the outside world. But remember, it's only exposed as read-only, because our objects are immutable! We'd have to construct another object if we want to "mutate" an existing item (IO monad aside).

The other thing to note is we don't consider functions as a part of our data type in the same way Java (or C++) does. A function is a function whether we define it along with our type or not. So we separate them syntactically from our type, which also contributes to conciseness.

Of course, we do have some notion of public and private items in Haskell. Instead of using the type defintion, we handle it with our module definitins. For instance, we might abstract constructors behind other functions. This allows extra features like validation checks. Here's how we can define our person type but hide it's true constructor:

```
module Person (Person, mkPerson) where

-- We do NOT export the `Person` constructor!
--
-- To do that, we would use:
-- module Person (Person(Person)) where
--   OR
-- module Person (Person(..)) where

data Person = Person String String String Int String

mkPerson :: String -> String -> String -> Int -> String
  -> Either String Person
mkPerson = ...
```

Now anyone who uses our code has to use the mkPerson function. This lets us return an error if something is wrong!

PYTHON

As our last example in this article, here's a simple Python version of our data type.

```
class Person(object):

  def __init__(self, fn, ln, em, age, occ):
    self.firstName = fn
    self.lastName = ln
    self.email = em
    self.age = age
    self.occupation = occ
```

This definition is pretty compact. We can add functions to this class, or define them outside and pass the class as another variable. It's not as clean as Haskell, but much shorter than Java.

Now, Python has no notion of private member variables. Conventions exist, like using an underscore in front of "private" variable names. But you can't restrict their usage outside of your file, even through imports! This helps keep the type definition smaller. But it does make Python a little less flexible than other languages.

What Python does have is more flexibility in argument ordering. We can name our arugments as follows, allowing us to change the order we use to initialize our type. Then we can include default arguments (like None).

```
class Person(object):

  def __init__(self, fn=None, ln=None, em=None, age=None, occ=None):
    self.firstName = fn
    self.lastName = ln
    self.email = em
    self.age = age
    self.occupation = occ
```

```
# This person won't have a first name!
myPerson = Person(
             ln="Smith",
             age=25,
             em="msmith@gmail.com",
             occ="Lawyer")
```

This gives more flexibility. We can initialize our object in a lot more different ways. But it's also a bit dangerous. Now we don't necessarily know what fields are null when using our object. This can cause a lot of problems later. We'll explore this theme throughout this series when looking at Python data types and code.

JAVASCRIPT

We'll be making more references to Python throughout this series as we explore its syntax. Most of the observations we make about Python apply equally well to Javascript. In general, Javascript offers us flexibility in constructing objects. For instance, we can even extend objects with new fields once they're created. Javascript even naturalizes the concept of extending objects with functions. (This is possible in Python, but not as idiomatic).

A result of this though is that we have no guarantees about how which of our objects have which fields. We won't know for sure we'll get a good value from calling any given property. Even basic computations in Javascript can give results like NaN or undefined. In Haskell you can end up with undefined, but pretty much only if you assign that value yourself! And in Haskell, we're likely to see an immediate termination of the program if that happens. Javascript might percolate these bad values far up the stack. These can lead to strange computations elsewhere that don't crash our program but give weird output instead!

But the specifics of Javascript can change a lot with the framework you happen to be using. So we won't cite too many code examples in this series. Remember though, most of the observations we make with Python will apply.

CONCLUSION

So after comparing these methods, I much prefer using Haskell's way of defining data. It's clean and quick. We can associate functions with our type or not, and we can make fields private if we want. And that's just in the one-constructor case! We'll see how things get even more hairy for other languages when we add more constructors! Take a look at part 2 to see how things stack up!

If you've never programmed in Haskell, hopefully this series shows you why it's actually not too hard! Read our Liftoff Series or download our Beginners Checklist to get started!

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.

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!

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!

Type Families in Haskell

Welcome to the conclusion of our series on Haskell data types! We've gone over a lot of things in this series that demonstrated Haskell's simplicity. We compared Haskell against other languages where we saw more cumbersome syntax. In this final part, we'll see something a bit more complicated though. We'll do a quick exploration of the idea of type families. We'll start by tracing the evolution of some related type ideas, and then look at a quick example.

This is a beginner series, but the material in this final part will be a bit more complicated. If the code examples are confusing, it'll help to read our Monads Series first! But if you're just starting out, we've got plenty of other resources to help you out! Take a look at our Getting Started Checklist or our Liftoff Series!

You can follow along with these code examples in our Github Repository! Just take a look at the Type Families module!

DIFFERENT KINDS OF TYPE HOLES

In this series so far, we've seen a couple different ways to "plug in a hole", as far as a type or class definition goes. In the third part of this series we explored parametric types. These have type variables as part of their definition. We can view each type variable as a hole we need to fill in with another type.

Then in the fourth part, we explored the concept of typeclasses. For any instance of a typeclass, we're plugging in the holes of the function definitions of that class. We fill in each hole with an implementation of the function for that particular type.

In this last part, we're going to combine these ideas to get type families! A type family is an enhanced class where one or more of the "holes" we fill in is actually a type! This allows us to

associate different types with each other. The result is that we can write special kinds of polymorphic functions.

A BASIC LOGGER

First, here's a contrived example to use through this article. We want to have a logging typeclass. We'll call it MyLogger. We'll have two main functions in this class. We should be able to get all the messages in the log in chronological order. Then we should be able to log a new message, which will naturally affect the logger type. A first pass at this class might look like this:

```
class MyLogger logger where
  prevMessages :: logger -> [String]
  logString :: String -> logger -> logger
```

We can make a slight change that would use the State monad instead of passing the logger as an argument:

```
class MyLogger logger where
  prevMessages :: logger -> [String]
  logString :: String -> State logger ()
```

But this class is deficient in an important way. We won't be able to have any effects associated with our logging. What if we want to save the log message in a database, send it over network connection, or log it to the console? We could allow this, while still keeping prevMessages pure like so:

```
class MyLogger logger where
  prevMessages :: logger -> [String]
  logString :: String -> StateT logger IO ()
```

Now our logString function can use arbitrary effects. But this has the obvious downside that it forces us to introduce the IO monad places where we don't need it. If our logger doesn't need IO, we don't want it. So what do we do?

USING A MONAD

One place we can start is to make the logger itself the monad! Then getting the previous messages will be a simple matter of turning that function into an effect. And then we won't necessarily be bound to the State monad:

```
class (Monad m) => MyLoggerMonad m where
  prevMessages :: m [String]
  logString :: String -> m ()
```

But now suppose we want to give our user the flexibility to use something besides a list of strings as the "state" of the message system. Maybe they also want timestamps, or log file information. We want to tie this type to the monad itself, so we can use it in different function signatures. That is, we want to fill in a "hole" in our class instance with a particular type. How do we do this?

TYPE FAMILY BASICS

One answer is to make our class a type family. We do this with the type keyword in the class defintion. First, we need a few language pragmas to allow this:

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE TypeFamilies #-}
```

Now we'll make a type within our class that refers to the state we'll use. We have to describe the "kind" of the type with the definition. Since our state is an ordinary type that doesn't take a parameter, its kind is *. Here's what our definition looks like:

```
class (Monad m) => MyLoggerMonad m where
  type LogState m :: *
  retrieveState :: m (LogState m)
  logString :: String -> m ()
```

Instead of returning a list of strings all the time, retrieveState will produce whatever type we assign as the LogState. Since our state is more general now, we'll call the function retrieveState instead of prevMessages.

A SIMPLE INSTANCE

Now that we have our class, let's make a monad that implements it! Our first example will be simple, wrapping a list of strings with State, without using IO:

```
newtype ListWrapper a = ListWrapper (State [String] a)
  deriving (Functor, Applicative, Monad)
```

We'll assign [String] to be the stateful type. Then retrieving that is as simple as using get, and adding a message will push it at the head of the list.

```
instance MyLoggerMonad ListWrapper where
  type LogState ListWrapper = [String]
  retrieveState = ListWrapper get
  logString msg = ListWrapper $ do
    prev <- get
    put (msg : prev)
```

A function using this monad could all the logString function, and retrieve its state:

```
produceStringsList :: ListWrapper [String]
produceStringsList = do
  logString "Hello"
  logString "World"
  retrieveState
```

To run this monadic action, we'd have to get back to the basics of using the State monad. (Again, our Monads series explains those details in more depth). But at the end of the day we can produce a pure list of strings.

```
listWrapper :: [String]
listWrapper = runListWrapper produceStringsList
```

```
runListWrapper :: ListWrapper a -> a
runListWrapper (ListWrapper action) = evalState action []
```

USING IO IN OUR INSTANCES

Now we can make a version a couple different versions of this logger that actually use IO. In our first example, we'll use a map instead of a list as our "state". Each new message will have a timestamp associated with it, and this will require IO. When we log a string, we'll get the current time and store the string in the map with that time.

```
type TimeMsgMap = M.Map UTCTime String
newtype StampedMessages a = StampedMessages (StateT TimeMsgMap IO a)
  deriving (Functor, Applicative, Monad)

instance MyLoggerMonad StampedMessages where
  type LogState StampedMessages = TimeMsgMap
  retrieveState = StampedMessages get
  logString msg = StampedMessages $ do
    ts <- lift getCurrentTime
    lift (print ts)
    prev <- get
    put (M.insert ts msg prev)
```

And then we can make another version of this that logs the messages in a file. The monad will use ReaderT to track the name of the file, and it will open it whenever it needs to log a message or produce more output:

```
newtype FileLogger a = FileLogger (ReaderT FilePath IO a)
  deriving (Functor, Applicative, Monad)

instance MyLoggerMonad FileLogger where
  type LogState FileLogger = [String]
  retrieveState = FileLogger $ do
    fp <- ask
    (reverse . lines) <$> lift (readFile fp)
  logString msg = FileLogger $ do
```

```
    lift (putStrLn msg)                -- Print message
    fp <- ask                          -- Retrieve log file
    lift (appendFile fp (msg ++ "\n")) -- Add new message
```

We can also use the IO to print our message to the console while we're at it.

#USING OUR LOGGER By defining our class like this, we can now write a polymorphic function that will work with any of our loggers! Once we apply the constraint in our signature, we can use the LogState as another type in our signature!

```
useAnyLogger :: (MyLoggerMonad m) => m (LogState m)
useAnyLogger = do
  logString "Hello"
  logString "World"
  logString "!"
  retrieveState

runListGeneric :: [String]
runListGeneric = runListWrapper useAnyLogger

runStampGeneric :: IO TimeMsgMap
runStampGeneric = runStampWrapper useAnyLogger
```

This is awesome because our code is now abstracted away from the needed effects. We could call this with or without the IO monad.

COMPARING TO OTHER LANGUAGES

When it comes to effects, Haskell's type system often makes it more difficult to use than other languages. Arbitrary effects can happen anywhere in Java or Python. Because of this, we don't have to worry about matching up effects with types.

But let's not forget about the benefits of Haskell's effect system! For all parts of our code, we know what effects we can use. This lets us determine at compile time where certain problems can arise.

Type families give us the best of both worlds! They allow us to write polymorphic code that can work either with or without IO effects. This is really cool, especially whenever you want to have different setups for testing and development.

Haskell is a clear winner when it comes to associating types with one another and applying compile-time constraints on these relationships. In C++ it is possible to get this functionality, but the syntax is very painful and out of the ordinary. In Haskell, type families are a complex topic to understand. But once you've wrapped your ahead around the concept, the syntax is actually fairly intuitive. It springs naturally from the existing mechanisms for typeclasses, and this is a big plus.

CONCLUSION

That's all for our series on Haskell's data system! We've now seen a wide range of elements, from the simple to the complex. We compared Haskell against other languages. Again, the simplicity with which one can declare data in Haskell and use it polymorphically was a key selling point for me!

Hopefully this series has inspired you to get started with Haskell if you haven't already! Download our Getting Started Checklist or read our Liftoff Series to get going!

And don't forget to try this code out for yourself on Github! Take a look at the Type Families module for the code from this part!