Welcome

This is a talk about Scala

Hi.

Thanks for coming along to this session.

My name is Richard, I work for a Scala consultancy called Underscore.

This talk is about how types help us.

and it's in the context of the Scala.

Scala's been around 12 years.

So no excuses: you must have seen it by now, tried it by now, or be using it already.

I'm going to just remind you of some of the features,

And point out how they let us use types to solve problems.

Scala has a powerful modern type system.

But you might have heard the downside to that.

That idea that it's complicated.

Negative

And it manifests itself in a number of ways. I quite like the way it's phrased here.

Where in the first year we dive in and love the conciseness of the language and get into the power, but ... then realise what you've written needs to be a bit simpler.

I find that a bit strange because it doesn't have to be like that.

Now, no language is perfect, all languages have their hard parts, but it seems like Scala has been in the spotlight over this recently.

When we're talking about getting a lot out of the type system, we have to also address this kind of concern.

The flip side of that is all about the benefits.

Positive

You'll see people talking development in terms like this.
This idea of writing down a type signature, and following the type to an implementation.
That sounds almost mystical.

"Follow the types"

This is definitely something I identify with.

And I want to get across the idea of why we bother, what the pay off is, for using types.

Ideas

We have these two themes.

1. We'll be looking at straightforward development with Scala.

2. And the ideas of what types can really do for us.

It may seem there a little bit of tensions between these ideas. But I see it as a progression.

Agenda

We'll deal with these ideas in three parts.

I'll start with some Scala, looking at patterns for simple Scala. We won't spend too much time there.

We'll move on from that and take a look at a really simple functional idea, encoding it as a type, and what it gets us.

And in the final part I'll try to share an idea of how this progresses into typelevel programming, where the compiler is doings ome of our work for us.

I'll work through tree examples to try to make this as concrete as possible.
My focus for all of this is application development. The examples come from that world.

I'm presenting these as three distinct examples, although what we see in part 1, helps us in part 2; and what we see in part 2, helps us in part 3.

And at the heart of this is the idea of Scala as a scalable language, in the sense of it scaling with our needs.

PART 1

Why? - Go

Let's get going.

But Scala is an un-opinionated language, and that can lead to some tricky situations. and developers taking different routes through the features on offer.

There was a post at the end of last year about a team, they were debugging a production problem...

And they only had one issues: they had no idea what the code was doing.

They came across a strange symbol. Which they called the starship operator.

Someone said out loud "what the hell is that?".

And I think that's probably a fair reaction. And an unhappy position to be in.

We could shrug this off and say it's a team issue... why is that getting into production, where's the review...
those are important and valid points, but the issue being made in this article is someone can write some Scala that the rest of the team didn't understand.

That's not what we want. What we all want is... and what the post goes on to say is...

It's about having a maintainable code base where you can have people cross projects easily and get new hires up to speed rapidly.

We all want that.

Power

If you're running a team, then you want them to have access to all the best power tools - there's a lot of potential productivity there. But you also want code that people want to work with, that no-one is alienated.

What can we do about that?

What can we do?

One thing is to be opinionated.

Here are 6 core Scala features that we think are the ones you need to know for everyday programming.

These are productive features.

There are more things in Scala, but we don't use them very often.

Other features are important for getting to the full power, but in an application... don't use them so much.

Some organizations already are opinionated. They know what they like, and that's what they use, and they have a way to introduce new ideas.

For example, ING bank have training activities for engineers. They've take education into their own hands, with class room activities, and sharing knowledge around. And mostly they are taking their Java developers into Scala, starting with syntax, and then into functional programming concepts. And that's working for them for hundreds of developers.

I don't know if they use these 6 ideas or not, but these are the 6 we use.

Two

I've picked out these two to talk about,

because I think it's maybe not a style much used in the Java world.

So hopefully will give you an idea of what I mean by straightforward. So maybe it'll give you something new to see.

But also these two are very common patterns we're going to use throughout this talk. And I hope you'll agree they lead to easily understood code, which means easy to maintain, also start to introduce types helping us.

ADT & SR

These might be unfamiliar terms to some of you.

Terminology can be daunting.
Functional programming feels like it has a lot of terminology,
but that's probably because functional programming are basically bandits, stealing all the good ideas from maths and logic.

Most of the common terms are pretty easy to work through.

I help run an functional programming meet up, and occasionally through some of these terms on a white board, unpick them, and see what they look like in different programming languages. You can get through four or so in an evening. It's good fun. You should try it.

Anyway... these terms...

They are handy labels, for amazingly useful idea.

Algebraic data types is all about data.

Structural recursion is about taking the data apart and doing something with it.

Let's deal with the Algebraic data types first.

This is all about modeling the world in terms of ANDs and ORs.

For example: a visitor to your web site is anonymous or logged in.
Notice the choice there. The or.

On the other hand, a logged in user is made up of two things.

an ID, and in the things we know about the user.

Notice the AND.

There are just these two patterns to algebraic datatypes: and or also have names, and collectively they are known as algebraic data type.

So what?

This is kind of modeling leads to a clear style of development.

Where the structure of the code we need to write, will follow the structure of the data.

Spec

So let's return to our specification. A visitor is anonymous or logged in.

In code that becomes a trait or abstract class. And two sub types.

The sealed part means the compiler will check that there are no other kinds of visitors for us to deal with.

Very handy as code evolves. If you introduce new types of visitor, the compiler will tell you if you've not handled them in the code base.

So we've translated our visitor is two kinds of things into code.

Spec 2

The second part of the specification is that logged in user has and ID and facts. And we've thrown in here that an anonymous user has an id.

Because we probably track them.

Now our code goes from this, to this encoding where we have the ORs.

You can still be anonymous or a user.

but we've no encoded the ands: A user has an id and a list of facts about them.

That's the data side of things.

SR

Now we need to do something with that data.
And that's the role of structure recursion.

As an example, perhaps we need to serve a advert for a visitor.

How to we implement this method?

We don't have to think to hard. We've done the thinking part in setting up the data.
We now need to deal with the two cases.

And in Scala we can do that with pattern matching. And don't confuse that with regular expressions. We're matching on the types and the structure inside the types.

serveAd

So we match on our visitor, and we know there are two cases to deal with.

This is two patterns: the first is the case of a User, we don't care what their ID is, but we are going to use whatever info we have about them.

The second pattern is the anonymous user case, and here we are picking up whatever id they have.

And we can produce some advert however we want. In the case of the user, maybe we find a relevant ad.
For an anonymous user, maybe we rotate based on their cookie id.

For the logged in user, we can use what we know about them -- their interests or preferences -- to serve up something useful.

And one thing to note here: these two methods -- relvantads and rotatedads - don't need to know anything about users.

They can we written in isolation from users, and just work on the data they need to work on.

Structure

One thing I love about this code, is the amount of structure I get.

If I was working in a dynamic language, maybe JavaScript, perhaps I'd just have an object with some properties in it. Rather than match on an anonymous user, I could stick my hand in that bag and rummage around for some property that would tell me it's an anonymous user. That's great, because I can throw in more properties without any ceremony.

I could write an IF here... if my user object has an anonymous flag, then I do one thing, otherwise if they have some facts, do something else... and superficially that wouldn't look to different from this code.

But there's a huge difference. Because I have this structure, I don't have to hold that distinction between visitors in my head.

I'm getting support from the types, and from the compiler, and that massively simplifies my job.

Summary

That kind of code, is pretty straightforward. You can be productive, and it occurs frequently, including in the standard library.

You have to decide to write code that way. You have to be opinionated in what you use, and that's how you avoid opening the code and saying "what the hell is that".

And we started to see how types start to help us, by having some structure to work with, but let's look at other ways they help.

PART 2

Set up

What do I mean by helping?

When I'm writing software, I want to make some progress.
And the recognition of a type can help me do that.
And when I see the idea, or type, I can automatically - almost mechanically - know what to do with that type.

In other words, it'll help me get on with my job.

That's the kind of help I mean.

Scenarios

To explain this, here are a list of things you might do when building an application.

Typically you need to be able to combine things together.
Bring two lists together and squish them into one.
Bring some statistics together and boil them down to something.
Gather messages and present them to a user.

We can approach all of these problems in terms of two things.

Combinable as a concept

And those two things are: a function for combining, and a value to represent nothing -- the starting or empty value.

What would that look like?

If you want to sum numbers, the combiner is addition and the starting point is zero.

If you want to combine text, you start with the empty string, and in Java and Scala use the + operator

Note that we not doing the same thing in each case. But the structure of the problem is the same. The fact that we use the + operation for numbers and the same symbol, +, for strings... that's not relevant here. That's not the similarity. The combine function can be kind of arbitrary, subject to some conditions we'll get to. The pattern is that there's empty value and way to combine two values.

Another example might help. If you have a set of things, you start with an empty set and union on to it.

And I've worked with a web framework that let you build up JavaScript expressions: starting with nothing, it used the ampersand symbol as a way to combine JavaScript statements into JavaScript programs.

The general pattern is is: for any type T, you need a zero for the T, and a way to bash two Ts together into one.

Monoid

So this is pattern that occurs.
And when we see patterns, we give them labels, or names
And this is referred to as a monoid.
There's another aspect to monids, the laws, that we get to.

The name monoid, it's a functional programming term, and as I mentioned earlier this is a concept from maths. Hence the weird and scary name.

It's another unfamiliar term, but hardly complicated.

There's only one time when Monoids are scary.
That's the 1966 Dr Who season that featured a creature called Monoid.

Example: list of numbers

So what? We've given a name to a simple thing. How does that help us?

Let's work through some examples, starting simply and building up to something bigger.

The boss says: those visitors we had, from earlier in the talk. What's the stats for the web site traffic?

If we know how many pages each visitor accessed, what's the total?

This is a trivial problem, right?
You know how to solve using a loop, or a library function.
You just have to add up the numbers.

Let's see what it looks like as a monoid instance.

For any T

The basic pattern we've seen is this: for any T , we need an empty and a combine.

And the interface for that could be this. This is two effectively abstract methods. One called called empty that must return a T whenever we implement it. And a function to combine two Ts into one T.

An actual instance of that could for addition could he this.
This is an instance of our Monoid interface. It works on Integers. The empty value is zero. That's what we need to start from for addition to work. And combining two values is adding them.

So we could use addition.empty in a loop, and addition.combine to bring values together in the loop.
That would be horrible. I might look like this.

And it's horrible because we're missing a theme that will recurr.
When you have an encoding of something, there's usual a general purpose function to go with it.

Just as we saw earlier: we can encode ADT, and that's mirrored by structural recursion for working on the data structure.
With monoids there's a general purpose function you're probably going to want to use.
And that function is called fold.

What is fold

Keeping with the theme of simplicity, we can write a fold function.

This is a version that's hard coded for addition. We will generalize it in a moment.

We have values in a list.
What we do here is match on our list of values,
And in the case of an empty list, Nil, the value to return is the empty value.
If, on the other hand, our list is made up of a value and the rest of the list, our result is that value and whatever we get from folding on the rest of the list. So it's a recursive algorithm.

Folding on list 1,2,3, will give us an answer of 6

Visually we can see that happening. Starting with a list of three values, we pick off the first value.
We add to that the result of folding the rest of the list.
We repeat the process.
And eventually we run out of values, so we use zero, and combine all those steps up. And they add up to 6.

Generalizing

That's OK, but hard coded. To generalize it we can drop in our monoid class. That's a parameter to fold, and we replace our hard coded values with empty and combine.

But we can go more general.

If we remove mention of Int, and say for some type T.... If we have a list of T and a monoid for those Ts, then... nothing else changes in the implementation.

Split

Notice, by the way, that m.empty is the only value we could have put in here. What we need to return is a T, and the only value we have of T in scope is empty. So imagine, sitting in your editor. And you've written up to here.

What can you put in place of the blue question marks? You need the value T. You've got an emty list, so that's no help. The only value you have is empty.

Some languages can figure that out for you. So Idris, for example, a totally different language, you could put your cusrsor after the arrow here, and hit ctrl alt S and it would solve the right hand side, and insert empty.

Do we like this code?

Although we've got to something more general, that's an illustration. I'd probably use an existing library for this that typically is still more general, not requiring a list either. The same idea can be used to "fold over" a vector, or a map, or a tree. And would be more efficient, and at least do tail recursive.

But the basic idea is this, and I wanted to show that it isn't a complicated idea

So what... we have a name for something simple, a monoid, and we can operate on it with a fold.

Sure, we can respond to the bosses request with a little bit of code.

That's not the benefit. We would have written that a bunch if ways.
So let's look at the benefits of why we might follow this route.

Benefits

Here are the main three benefits I see from us thinking in terms of monoids.

We get flexibility, we get to re-use monoids as they compose, and we get help problem solving.

Now composition, isn't something I'm going to talk about. The idea is broadly that you can use monoids as building blocks to get monoids for more complicated structures.

I want to focus on these other two.

Distinct

Functionality grows, because that's what functionality does, and we're asked about distinct visitors.

That is, of all the different visitors we get, we have various IDs, and we want to know how many different visitors we saw in some selection of the traffic.

How can we create that report? Well, one way is to use a Set. That is we can fold our visitors over a Set. What do we need to do that?

We need an empty value, and a way to combine values. Which we've seen already. We'd implement a monoid for sets, using union.

It's a one line change in the report.

Hyper

Or maybe the boss says: argh, we're out of memory.

Our list of visitors is too big here. What can we do?

There are algorithms that give you very good estimations. An example is HyperLogLog. This description doesn't explain much about what hyperloglog does, but you'll see it follows our monoid pattern. In some ways, great, we don't have to understand it. We can apply it, and it's a one line change to our code.

If you do want to know what the heck HyperLogLog is, there's a record you can find a nice talk from Papers we Love, on youtube.

An there are many useful algorithms like this. If the boss asks who are the most frequent visitors, we can use another approximating algorithm like count min sketch. Again the point here is to see the pattern, we're not going to go into the details. There's a talk from Laura at Twitter you can find at the scala exchange conference web site.

As an aide, those of you using Spark are maybe familiar with this already. If you're reducing any data, you're maybe using a monoid.

Parallel laws

What other flexibility do we get? We can safely apply our monoid in parallel.

Our counting or hyperloglog-ing, or looking for distinct visitors can all be safely split up into multiple jobs, and combined back into a single value. And we know that because of laws.

This is something else that functional programming has done for us. To be a monoid you have to have the structure I mentioned, but you also need to follow the monoid laws. And there are two laws. And the laws are really what make something a monid, I guess.

The laws look like this. For our addition monid we demand that applying empty to a value has no effect. And when adding, we demand that it doesn't matter how to group those operations. So 1 plus 2, when added to three, is the same as 1 added to the combined 2 and 3.

The same must apply for our string concatination monoid. Here our empty value is the empty string.

And this has to hold for any values. So any a plus the empty value is a. Any a plus b, then plus is c....

The's laws have names: identity and associativity
And more generally, in the language of the interface I've been using we say: a combined with empty is a, and a combined with b combined with c, is the same as....well, you get the idea.

There's no police enforcing that law. Typically, when you write an instance, you're write a test to go with it. Usually you'll do that with property based-testing, where you use a library that will generate edge cases and arbitrary values, and test your monoid.

And I think you can see that providing we obey those laws, our monoid - regardless of what it does -- will parallelize nicely. It doesn't matter if you do this in one thread of execution or many, you're going to get the same answer. And that's possible because the law of the data structure.

Thinking

Those benefits...

- Laws, so we know what the monid is defined as, allowing us to do things like run our computation safely in parallel.
- Being able to drop in other instances, because we can fold over our data using whichever instance we want
- Being able to compose instances

...They are great.

There's one more benefit I want to talk about. And this is more general in the sense that it helps guide us as we write software.

Let me give you an example.

Atom

There's a text editor called Atom.
It's a very nice open source editor from Github.

And there's a project called ENSIME which adds Scala support into various editors, including Atom.

And I wanted to add a feature to it. There's a bar at the bottom of the editor. Normally, it's open and shows you details of compiler error and warnings. But when it's closed like this, I want to see me this summary of errors.

In this project it's telling me about 10 errors.

To get that count, the way this works is ... the file is sent across the network to an interface for the compiler, and it sends back various messages. You don't get all the messages back in one go, they come back in a stream, which means you get answers quickly as they are found.

And sometimes you'll get a message saying " hold on, forgot those errors, I've found something else, and here are some other errors"

Does this sound familiar?

We have messages arriving that I want to combine into this summary.
And we have a reset, that takes us back to no errors and no warnings.

We can say: Its a monid. I know this!

And what that means... well it means we know we need a zero and a way to combine two values, and there are laws to obey...

But beyond that we also know what we're going do with those messages...we're going to fold over those messages.

This recognition of a pattern guides our next steps at the keyboard. It means, as a developer, I can progress my work.
I think that's a big part of adopting some of the more functional ideas.

And BTW, that was in CoffeeScript. The ideas in functional programming aren't fixed on a particular language.

Summary
We've looked at a very simple idea, and I've tried to give you an fel of how that actually helps us, in the real world, sitting at a keyboard. Sorry, stand at a keyboard.

There are many more ideas in functional programming. And there are patterns and laws behind them,

But there's only a handful that I find myself using

Despite functional programming, types, and indeed maths, being a big area to explore.

There's another way that types can help us.

PART 3

A taste of typelevel

Even though we're using straightforward parts of Scala, and only looked at a very simple type.

We've not used any exotic features, but we're also not excluded from using other parts of Scala when we need them.

In this last section I want to give you a taste of what that could be. And how the compiler can do some of our work for us.

I have two example to share. The first is to illustrate that we can push things we might usually view as runtime concerns down to the the compiler. And anytime we do that, the benefit is catching a problem when you press save in your editor, rather than when you run the code.

And we can write simple methods that take a list of headers, and a list of rows. And somehow turn that into CSV.

One thing that can go wrong is we don't have the right headers. Code changes over time, new fields get added to a report, and the column headers sometimes go missing. It's not the end of the world, not a huge deal.

What can we do to prevent that?

We can write a unit tests: good idea, nothing wrong with doing that.

We can add a check inside our csv method: some kind of assertion that headers and the rows should be the same length.

But that's not great as it'll give us an error at runtime. At least with our bug, we get a report at runtime.

Sized

Instead what we can do is ask the compiler to check the lengths match up.

Here's my CSV method. And as you'd expect the first argument is a list of strings, and the second argument is a list of rows, and each row is a list of strings.

The length of each row should match the length of the header.
The surprising thing is that this is possible with Scala. And it's possible because of some features that are called out as being complex, or undesirable.

But we we can benefit from those features via a library.

So in this case we can make use of a library, a library called shapeless.

The change to the code ... is something you'd take from the documentation. It puts a type parameter on csv, N, which had better be a natural number. Because we're going to be counting something.

And then our header and rows are wrapped in a Sized type, both of which share this N. In other words, the compiler is going to have to be happy that headers and rows both have this same N. The same length.

When we call CSV, we need a Sized collection, which we get from a constructor. So Sized Date there is constructing a List with a Size, or N of 1.
And because that doesn't match the length of my rows, it will be a compiler error.

I'm not showing you this to say: wow, go use this in your code. I'm showing it because it maybe a surprise that the Scala compiler can do these kinds of tricks. These kinds of things open my mind to the idea that maybe I can push more work down on the compiler.

You may find you have practical occasion to use this, but it's really only viable for small lists, and although this does produce a compiler error, it's a whopper. And although the information is there... it's exposing a lot of how the library is working.

The error follows the normal conventions. It's saying there's a type error. It found a size of natural number 1, but required a successor of 1 (or 2 as we often call it). And it tells you where it happened, but ... not a lot of fun wading through those.

But... what do you prefer: the compiler catching errors, or having runtime errors?

Let's look at another example.
One which is more useful and practical.

And one that has much better error messages, but is in the same area as capturing things at compile time.

Merge

It's reasonably common to have to compare data structures in some way. Merging database records with user data. And that can involve partial updates.

For example, our user class from earlier. Maybe we have these three fields: a database id, a name and an optional email address.

And we get updates to this record across the web. I'm showing some kind of patch request here. Just the name is being sent. But you can imagine different ways a user is supplying just one thing to update. Especially with ajax based interfaces where updates are being fired at a server as soon as you move out of a field.

Maybe what we want to do is represent an Update. The id won't update, but we might (optional) get an update to a name; and we might also get an email address update. Or possibly both.

What we want to get to is: taking a user, taking an update, and merging them sensibly. We preserve existing fields. We update fields we are told to update.

Scala lets you do that, and check the fields all match up at compile time. And it's not for these specific classes: it's can be done generically for any pair of classes. Without using run-time reflection or byte code hackery.

My colleague Dave Gurnell has put together a library, called bulletin. It's something like 60 lines of code.   But it's using some pretty advanced features.

And what does is provide a merge method, and this method type checks the fields of whateve case class it's given: so user and the update in this case. And it'll merge sensibly. This is really doing two things for us: it's saving us from a lot of boring field-by-field comparisons, but it's also helping us with code maintenance. If I make a change so the user and update classes mismatch in some way, we'll get a compiler error.

And it's a much nicer compiler error. So for example, if I rename user's name field to be family name, I'm told my user and update don't match.

If you're wondering how it can do that, well there are a couple of things. At the most abstract level, it's turning the user class and updated class into a list of the types. So user is a String for name, and optional string for email, and maybe there are other fields. And update is really a list of an optional string for name, and optional optional string for email.

And what happens is that the library splits this type apart into a head and the rest. And then the compiler has to go recursively figure out if the types match on the rest of the list of types. And this... if you recall... is the same pattern we used implementing fold. Split a list into the head and rest, recurse on the rest. Except in this case it's happening at the type level, with the work being done by the compiler.

But it'a also able to do that because Scala supports features like type constraints, implicit methods, allows us to create a hetrogenous list

So we need those features, but we don't need them when we write this.

Summary

The point here is to flag that the compiler can do more than maybe we usually let it do. And you may find situations where you benefit from that.

In the case of that sized collection, we're getting some compile-time assurances; in the case of the merging, we're also saving ourselves a fair bit of code without resorting to run-time reflection.

And if you decide to use libraries like this, or make use of the power features of Scala....

Go one of two ways

Well, that can go two ways.

You can end up with a surprise in the code, and someone saying: what the hell is this?

Or you can share the knowledge around, discuss what you want, and then the team can get a "It's a monoid, I know this" moment.

This brings us back round to the issue of Scala being complex. It has powerful features, and yes they are involved. But you can grow into them, if you need.

If we look back to 2008, we see Scala described like this: Scala stands for scalable language. It is designed to grow with the demands of its users.
That quote is from the first chapter of the *Programming in Scala* book.

The text goes on to talk about being able to write scripts, or complete programmes, but this idea of growing with your demands is one worth keeping in mind.

The picture I have is something like this: where you use the straightforward parts with types whenever you can, dip into the power features if you need it. But you have to share that with your team. We probably don't take enough time in our teams to talk enough about what we like and don't like. But if we do, sharing what we learn, things that are complex start to at least become familiar.

Summary

Keypoints

What have we seen?

We started with some very straightforward scala code. That's a style of code I use a lot, starting with writing down the types, and figuring out how I want to operate with them. There was nothing complicated there. The take away point: is have an opinion on what you use from the language.

We took a look at a very simple functional programming idea, the monoid.

What I'd like you to remember from that the ideas from functional programming help as your solving problems. They're not fancy ideas for annoying people with.

And finally, we took a brief look at some of the power features that Scala has. And you can ignore them, or start to dip into them as you need. If you're doing that, please make sure you share the learning around your team. That won't happen unless you create an environment for your team to share knowledge around.

On the first part, the straightforward part, the picture is set to get better. At the start of this year, Martin Odersky the father of Scala, set out his agenda to make the Scala simpler.

And on the second part, when it comes to making use of types, well I was using a library for that from a community called typelevel scala, which you can find here at typelevel.org. Theres a bunch of libraries there from foundational functional programming, to numerics things, and database libraries....many libraries. But what's good for us in particular is the focus on:

Functional Programming in Scala - not Scala trying to be another language like Java or Haskell, but using functional programming with Scala in idiomatic way.
And also making available learning resources, to make the ideas more accessible.

Both of those are goals. They are not done yet, but that's I think is the right direction to be going in. And I suggest you get involved in, try things, jump on gitter and ask questions.

Remember: Scala scales with the team, use the parts you want, you don't have to rush in and use it all. And make use of those type ideas from functional programming. They are there to help you. And share what you learn.

Thanks

Amanda
Wesley
Noel
Dave
Miles
Jono
Julio Capote
Alessandro Zoffoli