EDAF95/EDAN40 - Functional Programming Lecture Notes
L4: Types, Type Classes, and Data Structures.

Simon Kristoffersson Lind
simon.kristoffersson_lind@cs.lth.se

2024

In this lecture we will cover the basics of Haskell's type system. While doing so, we will cover *type inference*, Haskell's *basic types*, and how to create our own types. As fun/interesting example, I will alsow show some *type-level programming*.

1 Haskell's type system and type inference

In short a type system allows us to describe restrictions on variables by giving them a *type*. As an example consider the following code snippet:

```
1 a :: Bool
2 a = False
```

Here we create a variable `a`, and we tell the compiler that a *is a* `Bool`. In turn, this restricts us in how we use the variable `a`. Consider a function like so:

```
1 f x = x + 5
```

Here `f` uses the `+` operator to add 5 to its argument. Clearly, this operation is not defined for Booleans, and therefore we are not allowed to apply `f` to our variable `a`. Hence, we have restricted the ways in which we are allowed to use `a`.

So, why do we want types if they only restrict us? Well it turns out that types are good both for us, and the compiler. First of all, types help us

avoid **many** common errors, because the compiler can tell us "No, you're not allowed to do that with this type!". Second, types allow the compiler to know more about our variables, which in turn allows it to perform better optimizations. Faster code = better ☺.

Type inference is the process of automatically figuring out what types things are. In other words, we can write a code like this:

```
a = False
f x = x + 5
```

and the compiler will figure out the types:

```
a :: Bool
f :: Num a => a -> a
```

Neat! Though it's good practice to always write the types yourself.

Haskell implements a Hindley-Milner type system, which is *very powerful*. More specifically, it is *complete*, which essentially means that it can figure out the type of *anything* you throw at it. This has some interesting implications, which we'll see more of at the end of this lecture ☺.

2 Basic Types

Haskell includes a number of basic types, that are part of the language itself. The ones you'll see most commonly are `Bool`, `Char`, `String`, `Integer`, `Rational`, `List`, `Tuple`.

A Bool is just a Boolean, nothing fancy:

```
b :: Bool
b = False
```

The same goes for Char – it's just a character:

```
c :: Char
c = 'b'
```

Strings however are a bit more interesting. At first they seem just like Java (or many other languages):

```haskell
s :: String
s = "hello"
```

However, if you instead just write:

```haskell
s = "hello"
:type s
```

into the interpreter, you'll find that it says:

```haskell
s :: [Char]
```

Huh? Turns out `String` is just an alias for a *list of Char*. A little further down, we'll learn how to create our own type aliases.

Numbers are where it gets a little more complicated. First of all we have Haskell's "default" integer type, just called `Integer`:

```haskell
i :: Integer
i = 5129837012983701273013727837182631982370192038192 3
```

`Integer` is a flexible data type, which effectively means that you can write *really big* numbers.
We also have fixed-width types, like `Int`:

```haskell
i :: Int
i = 5
```

If we were to write a number that's too big here, the compiler would warn us.

Similarly, we have different types of decimal numbers:

```haskell
f :: Float
f = 5.3

d :: Double
d = 5.31203981029381

r :: Rational
r = 5.12378198028388189246178261273912389012836481927938012389 0
```

`Float` and `Double` are your standard fixed-width floating-point numbers. `Rational` is flexible, just like `Integer`, which means you can get *really precise* numbers. In practice, `Rational` is implemented as a fraction of two

`Integers`.

We have lists and tuples, that allow you to create collections of other types:

```
l :: [Int]
l = [1, 2, 3, 4]

t :: (Integer, String)
t = (1, "Hello")
```

Finally, we have functions:

```
f1 :: Num a => a -> a
f1 x = x + 5
```

I would read this type `a -> a` is "`a` maps to `a`". You could also say that "`f` is a mapping from `a` to `a`". If you're more used to Java or other non-functional languages, you might be more inclined to say "`f` takes an `a` and returns an `a`".

`Num a =>` is what we call a *type constraint*. You can think of it like an interface in Java. So what we're saying here is that "the type `a` has to be a `Num`". In other words, since we're using the `+` operator, the argument `x` has to be a number for `f1` to be well-defined.

Type signatures start to become a bit more complicated when we bring in more than one parameter for a function:

```
f2 :: Num a => a -> a -> a
f2 x y = x + y
```

Just like before, I would read this as "`a` maps to `a` maps to `a`". But, if you're not used to this type of function signature, this can easily get confusing: "does it return an `a`, and then another `a`?" No. Really the way we should read this is like so:

```
f2 :: Num a => a -> (a -> a)
```

In other words, `f2` is really a function that takes one `a` argument, and *returns another function*. Fundamentally, all functions in Haskell take only a single argument. Any function with more than one argument, really just takes one argument and returns a new function. Therefore, we are allowed to to cool stuff like this:

```haskell
f3 :: Num a => a -> a
f3 = f2 5
```

Now we have created a new function `f3` by essentially saving the value 5 in the `x` argument for `f2`. Note that this is a *completely separate function*, and `f2` still exists in its original form. This concept is called *Currying*, and `f2` is called a *curried* function.

3 Creating our own types

By far the easiest way to create a type of you own is a type alias using the `type` keyword:

```haskell
type MySuperCoolString = [Char]
```

This creates a "new" type called `MySuperCoolString`. In reality, all we've created here is an *alias* for the type `[Char]`. `MySuperCoolString` is just another name for `[Char]`. The standard Prelude `String` type is defined exactly like this.

If we want our type to be new *for real*, then the most basic way we can accomplish this is by the `newtype` keyword:

```haskell
newtype MyEvenCoolerString = CoolString String
```

`CoolString` is called a *constructor*, and we have used `String` as a *field*. When we want to create a variable of this type we have to use this constructor:

```haskell
s = CoolString "Hello"
```

Functionally speaking, this variable `s` is a `String`. However, the compiler won't let you use it in functions that take a `String`, because the compiler considers it a `different type`, even though they are functionally identical.

There are however some caveats. You cannot expect to use `s` just like you would use a `String`. If you're used to non-functional programming, you can thing of `MyEvenCoolerString` as a class that *contains a* `String`. As such, in order to use `s` as a string, you have to unwrap it first:

```haskell
f :: MyEvenCoolerString -> MyEvenCoolerString
f (CoolString s) = CoolString (reverse s)
```

Finally, it is also possible to create a new type that contains anything:

```
newtype MyVeryOwnType a = MyType a
```

Now `a` can be any type, which allows me to create variables that contain `anything`. For example I can do:

```
x = MyType "Hello"
y = MyType 42
```

Now `x` has the type `MyVeryOwnType [Char]` and `y` has type `MyVeryOwnType Integer`. In other words, this works much like template types in Java or C++.

Next we'll look at the `data` keyword, which is a more powerful version of `newtype`. Now you may ask:

"Then, why should we use `newtype`, if `data` is more powerful?"

While `data` is more powerful, `newtype` is easier for the compiler to work with. In other words, it can do a better job compiling things for you. Therefore, it is good to use `newtype` when you can.

So, what's the difference? `newtype` can be used with `exactly one constructor with exactly one field`. Whereas `data` can have many constructor, each with many (or zero) fields.

As a first example, consider a simple data type to represent seasons:

```
data Season = Spring | Summer | Autumn | Winter
```

Here we have four different constructors, and neither of them have a field. We would create variables like so:

```
x = Summer
```

Next, maybe we want to store the average temperature along with the season:

```
data SeasonTemp t = SpringT t | SummerT t | AutumnT t | WinterT t
```

Again, `t` is like a template, so we can put anything in this field. We create variables like so:

```haskell
x = AutumnTemp 15
```

We're also allowed to have different fields for different constructors. Using this we can create a very simple `Optional` type inspired by Java:

```haskell
data Optional a = None | Some a
```

In fact this is almost exactly how the `Maybe` type is defined in the standard Prelude.

Now, to illustrate multiple fields, we might want to create a type to represent a mapping from one thing to another:

```haskell
data Mapping k v = Map k v
```

When we want to access the fields, we can destructure the type just like I shoed before with `newtype`:

```haskell
key :: (Mapping k v) -> k
key (Map a b) = a

val :: (Mapping k v) -> v
val (Map a b) = b
```

Additionally, we can use so-called *record syntax* to give names to our fields:

```haskell
data FancyMapping k v = FancyMap { fancy_key :: k, fancy_val :: v }
```

This will give us the functions `fancy_key` and `fancy_val` for free. And they work exactly like the `key` and `val` functions we defined ourselves earlier.

It is also possible to use record syntax with `newtype`, as long as you stick to the rule of having only one field.

4 Example: binary search tree

Let's look at a practical example: creating a binary search tree.

First and foremost, let's define our data type:

```haskell
data Tree a = EmptyTree | Node (Tree a) a (Tree a)
```

Simple, right?
Basically, we're saying that a `Tree` is either `Empty`, or it contains an element of type `a` along with a left and right subtree.

But, hang on.
This is a recursive type definition, is that really allowed?
Yes, yes it is ☺.

Next, I suppose we want to implement an `insert` function, so we can add things to our tree:

```haskell
insert :: Ord a => (Tree a) -> a -> (Tree a)
insert EmptyTree x = Node EmptyTree x EmptyTree
insert (Node left val right) x
    | x < val = Node (insert left x) val right
    | x > val = Node left val (insert right x)
    | otherwise = Node left val right
```

Okay, let's break this down.

First, we simply state the type of this insert function, where we say that the type `a` has to be `Ord`. This is the requirement for us to use the comparison operators `<` and `>`. And we're also saying that `f` takes a `Tree`, then an `a`, and returns a `Tree`. Basically, the first argument is our tree, the second argument is the thing we want to add, and the return value is a new tree.
Second, we define our base case, when we're adding something to an empty tree. In this case, all we do is return a `Node` with the element inserted, and with both the left and right subtrees empty.

On the third line, we define our general case, when the tree is not empty. Here we deconstruct the tree argument, so that we can access its individual fields.

Finally, we create the three cases for insertion. If the new item `x` is smaller than our current value `val`, then we insert into the left subtree. If `x` is greater

than our `val` we insert into our right subtree. The only remaining choice, denoted by `otherwise`, is that `x` is exactly equal to our `val`, in which case we simply return our tree as it was.

A `contains` function can be implemented with similar ease:

```
contains :: Ord a => (Tree a) -> a -> Bool
contains EmptyTree x = False
contains (Node left val right) x
    | x < val = contains left x
    | x > val = contains right x
    | otherwise = True
```

Now we're ready to use our tree:

```
x0 = EmptyTree
x1 = insert x0 5
x2 = insert x1 3
x3 = insert x2 8
x4 = insert x3 10
x5 = insert x4 5 -- This will do nothing, since 5 is already in the tree
x = insert x5 1

contains x 5 -- True
contains x 16 -- False
contains x 1 -- True
```

You may wonder why I define first `x0` and then `x1` and so on. Coming from non-functional languages, it is tempting to write:

```
x = insert x 5
```

However, Haskell is a mathematical language, which means that this is a recursive definition. Thanks to Haskell's lazy evaluation, it will happily allow you to write such expressions, but the moment you try to evaluate `x`, it will result in an infinite recursion.

Now we get to the final piece of the puzzle, when it comes to creating our own types: *typeclasses*. Interestingly, Haskell has chosen the keyword `class` for this, which can make it confusing when you're used to object-oriented languages. Really you should think of `class` in Haskell like an interface in Java, or an abstract class in C++. The word `class` in Haskell refers to the creation of a *class of types*, hence we call it a *typeclass*.

```haskell
class Set t where
    class_insert :: Ord a => t a -> a -> t a
    class_contains :: Ord a => t a -> a -> Bool
```

Here we define our typeclass called `Set`. For any class to be considered a set, it should implement an insert function, and a contains function, which is exactly what we describe here. Though I've called them `class_insert` and `class_contains` to highlight the fact that they're different from our previous `insert` and `contains` functions.

Our `Tree` data type obviously fulfills our requirements for being a `Set`, so we can tell the compiler that our tree *is a set*:

```haskell
instance Set Tree where
    class_insert = insert
    class_contains = contains
```

Since we've already defined our own `insert` and `contains` functions, we can simply reuse them here, It would also be allowed to define them directly in the `instance` definition.

This concludes the part of this lecture that is necessary for passing the course. What follows on the next few pages is just some fun extra content, for those who are interested ☺.

5 Haskell's type system is Turing complete

As I mentioned before, Haskell's type system is *complete*. A result of this, is that it is also Turing complete, which means that we can program things directly in the types.

Fun! ☺

However, by default GHC has a few safety-rules to ensure that it is able to compile the programs you write. So, in order to use the full capabilities of Haskell's type system, we need to turn some of those safety-rules off:

```
{-# OPTIONS_GHC -fno-warn-missing-methods #-}
{-# OPTIONS_GHC -fno-warn-simplifiable-class-constraints #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}
```

With that out of the way, let's do some type-level programming!

We "magically" create booleans:

```
data True
data False
```

These are just types, and we'll pretend that they have some meaning ☺.

We can use these to define operations on Booleans:

```
class Not b r | b -> r
instance Not False True
instance Not True  False
```

Here we're using some fancy syntax `b -> r`. While that may look like a function type, what it actually means here is that the type `r` (for "result") *is uniquelly defined* from the type `b`. In other words, we're telling the compiler "I'm not going to give you the type of `r`, but I'm going to give you all you need to figure it out".

By then defining exactly two instances, corresponding to `b = True` and `b = False`, we constrain the type of `r`.

Using the exact same principles, we can bring in two types to construct more complicated operations:

```
class Or b1 b2 r | b1 b2 -> r
instance Or True  True  True
instance Or True  False True
instance Or False True  True
instance Or False False False
```

Now, how do we use these to perform actual computations?

We're going to use a little trick to make the compiler extract the result type for us. First, we define a simple typeclass:

```
class Result r where
    result :: r
```

By creating an instance of this typeclass, we allow the compiler to infer the type of `r` for us. Then we'll be able to check the result by `:type result`.

So now all we have to do is define the instance, which makes the actual type computation happen:

```
instance (Or True False r) => Result r
```

Here we're saying "if the types `True False r` is a `Or`, then create instance `Result r`". Obviously, we haven't specified `r`, so the compiler will figure out that "okay, the only valid instance is `OR True False True`, so `r` must be `True`". From there it creates our instance `Result True`.

With that done, we can check our result:

```
:type result -- True
```

Now I should note that since we created a typeclass called `Result`, we're not allowed to create another one with the same name. In other words, if we want to perform more computations, we'll have to name them differently, for example:

```
class Result1 r where
    result1 :: r

class Result2 r where
    result2 :: r
```

Using the same tricks we can create numbers:

```
data Zero
data S n
```

Here we've defined a type called `Zero` to represent the number zero. Then we define another type `S n`, where `n` is another number type. Technically `n` can be annything, but we decide it shall be another number type.

These are called Peano numbers, and `S n` refers to the *successor* of `n`. In this system, the number one is represented by `S Zero`, the number two is represented by `S (S Zero)`, and so on...

We can use our Booleans to define equality operations. As an example, here I've implemented the LessThan operator (<):

```
class LessThan a b r | a b -> r
instance LessThan Zero  Zero  False
instance LessThan (S x) Zero  False
instance LessThan Zero  (S x) True
instance (LessThan a b r) => LessThan (S a) (S b) r
```

That last line should be read as "if `a`, `b`, and `r` is an instance, then `(S a) (S b) r` is also an instance". In other words, this creates a sort of infinite expansion of instances, which allows the compiler to figure out the `r` type for any two numbers.

And of course we can perform computations. Here I'm checking if $4 < 5$:

```
class Result r where
    result :: r

instance (LessThan (S (S (S (S Zero)))) (S (S (S (S (S Zero))))) r)
    => Result r

:type result -- True
```

The fun doesn't end there. We can create lists:

```
data Nil
data Cons x xs
```

Here we create a type `Nil` to represent an empty list, and a type `Cons x xs`. Just like the plain Haskell syntax for list, we use `x` to signify the head of the list, and `xs` to signify the rest of the list.

In other words, we can create a list of the types `Int`, `Float`, and `Double` like so:

```
Cons Int (Cons Float (Cons Double Nil))
```

And we can create an operation to get the head of the list:

```
class Head list x | list -> x
instance Head Nil Nil
instance Head (Cons x xs) x

class Result r where
    result :: r

instance (Head (Cons Int (Cons Float (Cons Double Nil))) r) => Result r

:type result -- Int
```

And we can create a Range operator that gives us a list of integers like $[3, 2, 1, 0]$

```
class Range n xs | n -> xs
instance Range Z Nil
instance (Range n xs) => Range (S n) (Cons n xs)

class Result r where
    result :: r

instance (Range (S (S (S (S Z))))))) r) => Result r

:type result
-- Cons (S (S (S Z))) (Cons (S (S Z)) Cons (S Z) (Cons Z Nil))
```

The possibilities are endless ☺.